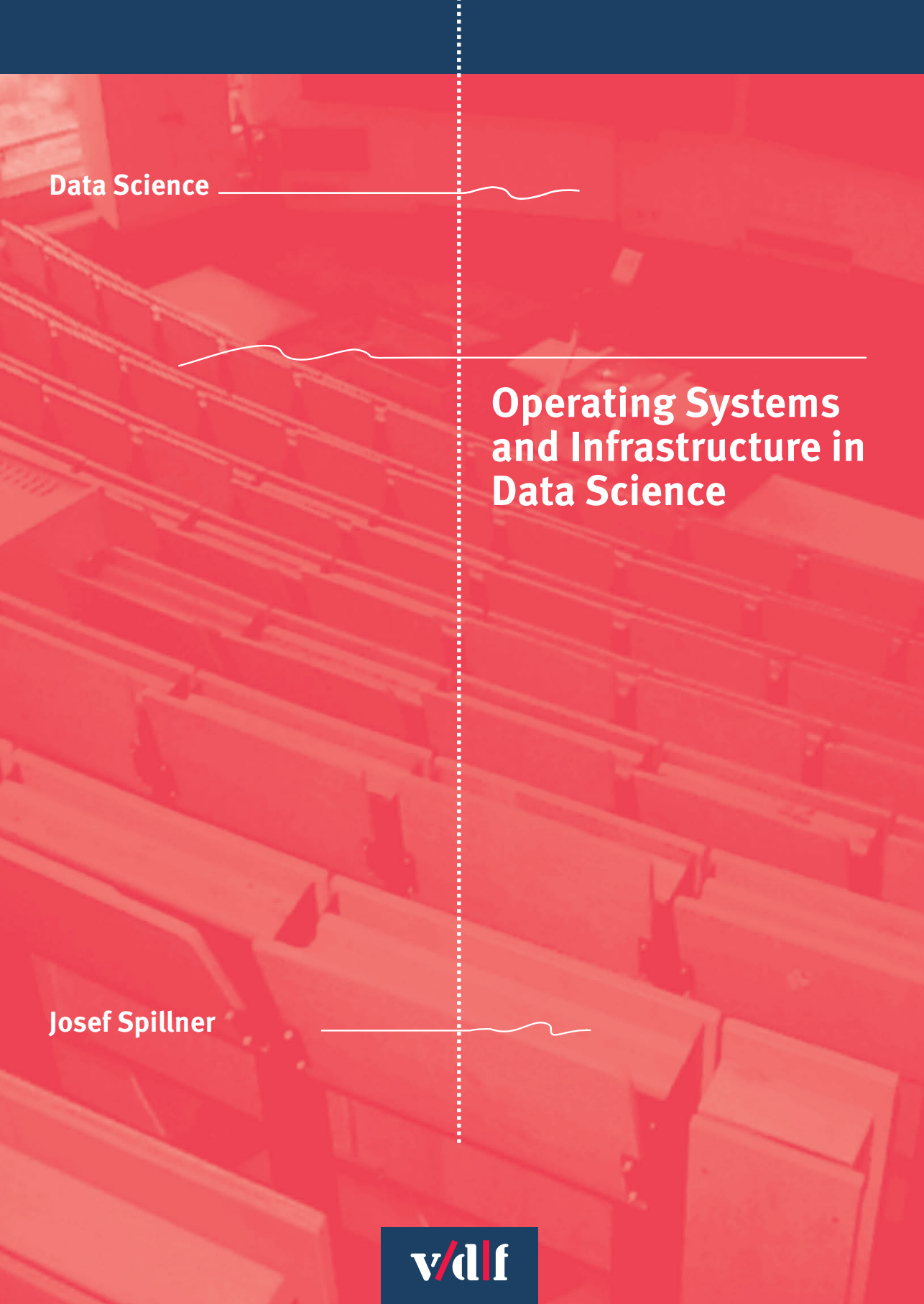
Data Science
Operating Systems and Infrastructure in Data Science
Josef Spillner
v/d/f

Operating Systems and Infrastructure in Data Science

1. Auflage

vdf Hochschulverlag AG
an der ETH Zürich

Josef Spillner

Operating Systems and Infrastructure in Data Science

1. Auflage

Financial support for the publication of this book was provided by the ZHAW University Library (HSB).

The author is affiliated with ZHAW.

Bibliografische Information Der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de abrufbar.

This work is licensed under creative commons licence CC BY NC SA.

ISBN 978-3-7281-4167-5 (Printausgabe)

Download open access:
ISBN 978-3-7281-4168-2 / DOI 10.3218/4168-2

1. Auflage 2023

© vdf Hochschulverlag AG an der ETH Zürich

www.vdf.ethz.ch
verlag@vdf.ethz.ch

Contents

1 Introduction **9**
- 1.1 Prerequisites . . . 10
- 1.2 Target competences and anticipated skillset . . . 10
- 1.3 Book structure . . . 11
- 1.4 Dedication . . . 11

2 Concepts: Programming, Data Representation and DataOps **13**
- 2.1 Data Structures: Graphs, Streams and Units . . . 13
  - 2.1.1 Graphs . . . 14
  - 2.1.2 Streams . . . 15
  - 2.1.3 Units for Data and Resources . . . 15
- 2.2 Data Formats . . . 16
- 2.3 Compute-Centric Processing: Pipelines and Workflows . . . 17
- 2.4 Data-Centric Processing: Sharding and Map-Reduce . . . 19
- 2.5 Event Processing, Handlers and Hooks . . . 19
- 2.6 Encapsulation: Functions, Tools, Containers and Services . . . 20
- 2.7 Data Management, Engineering and Operations . . . 21
  - 2.7.1 Data Engineering . . . 22
  - 2.7.2 Data Integration . . . 23
  - 2.7.3 DataOps . . . 23
  - 2.7.4 Reproducibility . . . 24

Repetition . . . 25

3 Concepts: Operating Systems **27**
- 3.1 Fundamentals . . . 27
- 3.2 Current Operating Systems . . . 29
- 3.3 Building Blocks: Executables, Processes and Resource Management . . . 30
- 3.4 Isolation, Virtualisation and Containerisation . . . 33
- 3.5 File System, Paths and File Access . . . 34
- 3.6 Networking . . . 36

3.7 User Management, Authentication, Authorisation and Credentials 38
Repetition 39

4 Concepts: Infrastructure 41
4.1 Networks and Internet 41
4.2 Networked Computers 43
4.3 Services and Platforms 43
4.4 Parallel and High-Performance Computing 45
4.5 Cloud Computing 47
4.5.1 Full application hosting 48
4.5.2 Partial hosting and on-demand offloading 48
4.5.3 Cloud backup 49
Repetition 49

5 Applications and Tools 51
5.1 Fundamentals 51
5.2 Mastering Tools 53
5.2.1 Text-mode interaction 53
5.2.2 Types of tools 54
5.3 Shells 54
5.3.1 Overview on shells and terminals 55
5.3.2 Local shell access with Bash 55
5.3.3 Bash variables 57
5.3.4 Bash commands 58
5.3.5 Remote shell access with OpenSSH 62
5.3.6 Advanced shell management with Screen and TMux . . 64
5.4 Useful shell tools 65
5.4.1 Hardware resources exploration 66
5.4.2 Operating system exploration 67
5.4.3 Time- and event-related commands 70
5.4.4 Managing data in files and directories 72
5.4.5 Creating, viewing and editing files 76
5.4.6 Networking 78
5.4.7 System administration 79
5.5 Shell programming 84
5.5.1 Vocabulary and interaction with scripts 84
5.5.2 Job management 86
5.5.3 Control flow programming 87
5.5.4 Shell functions definition 88
5.6 Python modules for OS interaction 88
5.6.1 Running the Python interpreter 89
5.6.2 Modules 'os' and 'sys' 90
5.6.3 Module 'shutil' 92

5.6.4 Module 'tempfile' . . . 92
5.6.5 Module 'argparse' . . . 93
5.6.6 Module 'subprocess' . . . 94
5.6.7 Module 'socket' . . . 94
5.7 Package management . . . 95
5.7.1 Python package management with Pip . . . 96
5.7.2 Advanced Python package management with Pipx and Poetry . . . 97
5.7.3 Package management for other programming languages 98
5.7.4 System package management with APT . . . 100
5.8 Container management . . . 101
5.8.1 Introduction to Podman . . . 102
5.8.2 Fetching and running containers . . . 103
5.8.3 Building custom container images . . . 104
5.9 Data management and version control . . . 105
5.9.1 Delta synchronisation with RSync . . . 105
5.9.2 Version control with Git . . . 107
5.9.3 Basic usage of Git . . . 107
5.9.4 Advanced usage of Git . . . 109
5.10 Data processing tools . . . 111
5.10.1 Text search . . . 111
5.10.2 Text processing . . . 112
5.10.3 Numeric processing . . . 113
5.10.4 Media formats . . . 114
5.11 Structured data processing . . . 115
5.11.1 Format-specific processing . . . 115
5.11.2 Training and inference . . . 117

6 Middleware **119**
6.1 Programmatic data serving . . . 119
6.1.1 Third-party module 'flask' . . . 120
6.1.2 Third-party module 'streamlit' . . . 122
6.1.3 Third-party module 'bokeh' . . . 122
6.2 File system abstractions and network storage . . . 123
6.2.1 Basic FUSE operations . . . 124
6.2.2 Selected file systems and synchronisation . . . 125
6.3 Database interaction and management . . . 126
6.3.1 Embedded relational databases with SQlite . . . 126
6.3.2 Networked relational database systems . . . 127
6.3.3 Beyond relational databases . . . 128
6.4 Message brokers for real-time data processing . . . 129
6.5 Parallel and distributed computing . . . 130
6.5.1 Data processing with Spark . . . 131

6.6 Model serving . . . 133
6.6.1 BentoML model serving . . . 133
6.7 Data integration . . . 135
6.7.1 Meltano data integration . . . 135
6.8 Workflows and distributed scheduling . . . 137
6.8.1 Airflow task and workflow specification . . . 137

7 Collaboration and Governance Platforms 141
7.1 Scientific notebooks . . . 141
7.1.1 Jupyter notebooks . . . 142
7.1.2 Working with notebooks . . . 142
7.2 Code and data lifecycle management . . . 143
7.2.1 Gitlab as repository management platform . . . 144
7.2.2 Gitlab as delivery platform . . . 145
7.3 Data catalogues and governance . . . 146
7.3.1 ODD deployment . . . 146
Repetition . . . 147

8 Execution and Orchestration Platforms 149
8.1 Virtual machines management . . . 149
8.1.1 Using OpenStack web interface and API . . . 150
8.2 Container management . . . 152
8.2.1 Kubernetes ecosystem . . . 152
8.2.2 Kubernetes installation . . . 153
8.2.3 Working with Kubernetes . . . 154
8.3 Cloud services . . . 155
Repetition . . . 155

9 Global Infrastructure 157
9.1 Data pipeline infrastructure . . . 158
9.1.1 OpenTransportData . . . 158
9.1.2 Renku Lab . . . 159
9.1.3 Zenodo . . . 161
9.2 Distributed applications infrastructure . . . 161
9.2.1 Etcd Discovery . . . 162
9.2.2 Dweet . . . 162
Repetition . . . 163

Solutions 165

Appendix (only provided for the printed book edition)
Complex Tasks and Exercises
Electronic Resources

Chapter 1

Introduction

The advent of the Stone Age around 2.6 million years ago marked a small but significant jump in human civilisation. In the Stone Age, humans started to become more productive through long-lasting and re-usable stone tools, first in the form of pebble tools and (much) later in the form of hand axes.

Fast-forward to several industrial revolutions that have defined and shaped modern highly productive civilisations over the past centuries. From agriculture over production to automated services and finally into the digital work domain, mastering tools has become increasingly essential skills to be productive. An interesting quote, *A man is only as good as his tools*, is attributed to Emmert Wolf, widely considered to have lived over a hundred years ago but without any further trace of existence. The statement still qualifies as valid, although with notable changes. First, while in traditional industries biological gender differences still play a role today, in digital work such as data science they do not. Second, individual work still remains important, but certain efforts require teamwork with various flavours of collaboration and coordination. Hence, the newer quote *A tool is only as good as the team using it* by Matthew Stublefield in 2019 gives the right educational context for this book.

In data science, mastering a system environment with its tools and processes is essential to achieve minimum productivity. Feeling alien to an environment, using the wrong tools or combining the right tools in the wrong order can lead not only to effectivity limitations but also yield wrong results. Hence, in this book, besides basic computer knowledge and programming skills, students on data science are empowered to assemble a battery of useful tools to employ in the right situation, ranging from small, versatile command-line tools to powerful online platforms. Compared to mastering a single programming language and thus controlling an application logic *in the small*, something that can be fitted into a few functions on the screen, this book advances the skills to *programming in the large*, beyond the boundaries of individual processes or

machines. Programming in the large means defining and orchestrating complex data-centric processes involving multiple tools, platforms and resources. The eventual goal for the reader is thus to be able to define data, model and code that should be provisioned and monitored as services in appropriate distributed infrastructures – from hosting data and models to running software in the cloud. As studying is only the first step towards practical application of skills in a professional setting, this book should therefore be a good starting point for students of data science and computer science, digital life sciences, digital mobility and similar curricula.

1.1 Prerequisites

The reader is expected to bring basic programming skills in an imperative language. This encompasses the definition of variables and control loops as well as the declaration and invocation of functions, including constructor methods to instantiate objects of predefined classes. To keep matters simple, this book assumes knowledge of the Python programming language and occasionally considers the Python interfaces to operating systems functionality. However, readers with knowledge in another language can also find their way around the explanations and exercises.

The reader also needs the ability to fully exploit the keyboard. If typing special characters beyond alphanumeric signs, such as `(@):$^`, poses any challenge, the advice is to train that before reading on.

1.2 Target competences and anticipated skillset

By ingesting the content in this book, including active participation in the repetition parts, the reader can expect to achieve learning goals on the three lower taxonomy levels: knowledge, understanding and application. Moreover, the avid reader is put in the position to decompose larger problems into smaller ones and thereby compose smaller tools into pipelines, workflows and other relationship models to address the larger problem as a whole, either fully or in part.

Eventually, the reader shall be brought into a position to fully control a computer, and even a distributed computer network, and make it work towards the processing of data. Such skills are indispensable to maintain the digital souvereignty on a personal level and in a business context. Instead of having to pay for data hosting and analytics services, it is certainly benefitial to at least maintain the option to do all of this oneself, and moreover, even when using such services, to fully understand what is happening and what could be improved.

1.3 Book structure

The book is structured into the following main parts, reflected as chapters with either a conceptual or practical focus. First, advanced programming concepts such as workflows with pervasive use across operating systems and data science infrastructures are introduced along with data concepts. On a technical-conceptual level, operating systems are then explained in greater detail for their influence on how users operate tools and how tools access data and interact. Next, applications and tools are introduced for use in local and networked environments, with emphasis on their practical use to solve smaller tasks. This chapter is followed by one on middleware as well as one more on collaborative platforms that combine middleware and tools. More complex data science infrastructures that combine cloud facilities with the aforementioned tools, middleware, platforms and workflows are then explained. Finally, non-commercial global platforms that make sense to be simply used at least in the exploration and prototyping stage of data science projects are described.

A glossary is not provided. Nowadays, there is no shortage of online resources to dive deeper into specific terms and topics whenever necessary. And yet, technology terminology is prone to ambiguity at times. Consider the word *key*. In hardware, it refers to a plastic key on the keyboard. In security discussions, it refers to a unique value or a unique sequence of bytes. In programming, it refers to the index that is uniquely associated to a value. All three semantic interpretations are present in the book in multiple occurrences. The book takes some effort to make these distinctions clear, but terms might be used colloquially at times, so that careful and conscious reading is recommended.

Command syntax that can be typed for execution is set in `teletype` font. For technical reasons, longer commands are hardcut on the right column edge.

1.4 Dedication

Operating systems and infrastructures have developed over time with millions of lines of code. None of that would be possible without the countless hours invested by free and open source software developers, community activists, researchers and skilled engineers. There are people behind the software tools introduced in this book, and this becomes apparent when they leave us. Bram Moolenaar, Sven Guckes and Dan Kohn are three shining examples whom the author had the pleasure to meet and to discuss with. And they would certainly like it if more people became proficient and self-determined on the command line to build solutions for all data science challenges.

Chapter 2

Concepts: Programming, Data Representation and DataOps

Programming in the large is conceptually not much different from programming in the small, but the terms and technologies differ, as do the implications of unsuitable algorithms, incorrect code or wrong administrative processes. In this section, a few terms are briefly explained, so that the references to them in tool and platform explanations later become clear: graphs and streams, data pipelines, workflows, shards, event processing, microservices and several others; and eventually, administrative concerns around the term DataOps.

2.1 Data Structures: Graphs, Streams and Units

In programming languages, native data types encompass scalar types (bytes, integers, floating point numbers, boolean values), vector types (lists, tuples, strings, sets, byte arrays), associative types (dictionaries) and geospatial types (dates, coordinates). Data represented in these types are always finite and lack arbitrary references. In practice, more types of data exist. Two important data structures indispensible for expressing data-driven activities are introduced here: graphs and streams. While there are niche languages for them such as Streams Processing Language (SPL), they are handled through libraries in mainstream languages. Their introduction here is therefore given broadly to understand them in every context. The section concludes with a summary view on physical units to express larger quantities in data and resources.

2.1.1 Graphs

Graphs consist of nodes (vertices) and edges – $G = (V, E)$ and are often expressed with graphical notation for human users. The edges can be undirected, unidirectional or bidirectional. As graph-processing algorithms take a predefined or arbitrary starting point and then descend into the graph iteratively or recursively, endless cycles might result from traversing the edges. The most useful graphs are, however, constrained to be cycle-free and (single-)directed, making them elements of the practical subclass of Directed Acyclic Graphs (DAGs). In DAGs, each node may be a splitting node with multiple outgoing edges or a joining node with multiple incoming edges. Moreover, both edges and nodes might be weighted, coloured and otherwise given attributes that are then taken into account in graph-processing applications.

Standard algorithms for graph structures encompass the calculation of the shortest path, or the path with lowest or highest sum of weights, between two nodes and the determination of all nodes reachable from a given node with less than a defined number of hops. Graph rewriting is the process of optimising a given graph structure by eliminating redundancies or introducing controlled parallelism or encapsulated subgraphs.

An example for a graph data structure and its application in the business data domain is the list of stations and the connectivity links between them in General Transit Feed Specification (GTFS), a tabular open format specification for public transport networks.[1] Transitive connections with stopovers can be trivially calculated based on the graph structure. Another example in agriculture is given by the relationships between producers and consumers of manure. Some farms produce a lot and have many associated consumers, with the volume of manure expressed as edge weights, while others may self-consume everything and thus remain isolated nodes instead of being on the graph.

Trees are subsets of graphs with only splitting and no joining nodes, i.e. for each node there can be multiple outgoing edges but only one incoming edge. Nested trees are therefore useful to represent arbitrarily nested hierarchies. Sequences are in turn subsets of graphs without any splitting nodes, and often with implied chronological order. Fig. 2.1 compares these graph structures with representative examples.

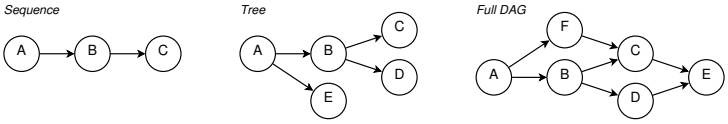

Figure 2.1: Comparison of graph structures

[1] GTFS example: `https://opendata.swiss/de/dataset/timetable-2023-gtfs2020`

2.1.2 Streams

Streams are (potentially) endless data structures that allow for incremental processing. Streams may be represented as characters, lists, trees or other structures, especially those permitting partial representation, which is the case for lists. A stream can be read or written sequentially but not at random locations. Furthermore, while already read data can be buffered to some extent, older data may be completely lost. Hence, in simple stream processing, data records are processed individually or in microbatches. In complex stream processing, operators aggregate information with modest size, which can be kept as opposed to the raw underlying data, which are transient.

2.1.3 Units for Data and Resources

The atomic unit for data is a byte, divided into 8 bits, each assuming the value 0 or 1. Hence, a byte can represent $2^8 = 256$ bits. This is tiny amount of data for today's system, hence more human-friendly prefixes have been introduced for larger amounts of data. Due to historic development, there are both the *Système international d'unités* (SI) prefixes known from physics on the basis of 1000 and the prefixes based on the binary system, more accurately on the basis of 1024. What causes additional confusion is that, in colloquial communication, sometimes one is meant but the other is referred to.

Modern 64-bit processors can hold up data with a volume of up to 8 bytes (64 bits) natively in their registers, corresponding to a `long` value and twice the size of an `integer` value. Anything larger than that, especially arrays such as strings, can only be natively represented in main memory and on disk, and this is where the prefixed units become relevant.

On the SI scale, $10^3 = 1000$ bytes are one kilobyte (KB), $10^6 = 1000000$ bytes are one megabyte (MB), $10^9 = 1000000000$ bytes are one gigabyte (GB), and $10^{12} = 1000000000000$ bytes are one terabyte (TB). The largest consumer-level hard disks nowadays have a capacity of few TBs, making this the largest prefix with common practical use. On the binary scale, $1024^1 = 1024$ bytes are one kibibyte (KiB), $1024^2 = 1048576$ bytes are one mebibyte (MiB), $1024^3 = 1073741824$ bytes are one gibibyte (GiB), and $1024^4 = 1099511627776$ bytes are one tebibyte (TiB). As can be noticed from the numbers, the deviation to the SI units increases with the order of magnitude with approximately 2.4%, 4.9%, 7.3% and 10.0%, which is why the distinction has become more important over time.

Storage resources such as HDDs and SSDs on a computer are typically measured in SI units, whereas main memory is measured on the binary scale. Compute resources are measured by fractions of their capacity, for example, a tenth of a CPU core (or 100 milli-CPUs), over time.

2.2 Data Formats

Within program code, native datatypes and their compositions (for instance, linked lists or graphs) are suitable to represent the structure and content of information. However, as soon as data need to be transferred to other programs or to the outside world, such structures need to be serialised in appropriate formats. The main structures for larger quantities of data are tabular and tree/graph data. Most programming languages have their own specific serialisation formats, such as Pickle in Python, that promise high performance but reduce the ability to exchange such data with software written in different languages. To increase interoperability, standardised textual representations exist for both tabular and tree structures, whereas there are only application-specific formats for most graph data.

Tabular data can be represent in text as CSV (Comma-Separated Values). All rows correspond to a line of text, whereas all columns correspond to tokens per line. These tokens are either unquoted (e.g. `house`) or quoted (`"green house"`), and separated by a comma or by another typical separator sign (`,;|`) or a tabulator (⇄). CSV files may have a header line with column names, but sometimes these names are application-defined and not represented in the data. Hence, overall CSV is a loosely defined format, leading to many interoperability challenges but with good support for line-based stream processing.

XML (eXtensible Markup Language), JSON (JavaScript Object Notation) and YAML (Yet Another Markup Language) are all capable of expressing tree-structured data beyond tables. To represent the hierarchical structures, these formats use different syntax, such as angle brackets (`<>`) in XML, lists and dictionaries in JSON (`[]`, `{}`) and indentation in YAML. Several derived formats exist and are standardised, such as JSON Patch to represent differences between two JSON files again as JSON document. While stream processing exists for these formats, this is not the norm for many applications. Rather, a full file is typically deserialised into memory.

The following listing compares three out of these four structured data formats: CSV, JSON and XML. Typing information is more explicit in JSON but across all formats requires an external schema to avoid heuristic determination.

```
CSV: person;age     JSON: [{"person": "Hans", "age": 22}]
     Hans;21        XML:  <persons><person name="Hans"><age>
     Heidi;22              22</age></person></persons>
```

To inform about the data format of any given document without having to inspect it, media type specifications such as Multipurpose Internet Mail Extensions (MIME) introduced a classification system. Accordingly, the textual representations of tabular and tree-structured data can be expressed as `text/csv`, `text/xml`, and (not fully consistent) `application/json` and `application/yaml`.

Schemas may be defined especially for JSON and XML, unsurprisingly called JSON Schema and XML Schema, respectively. A data schema informs about mandatory and optional fields, their names and data types, permissible value ranges and further constraints. For instance, the regular reporting about COVID-19 case numbers follows a strictly defined format.[2]

The concrete byte-level format of the data depends on its machine representation, which may be subject to further transformations. These include encoding, compression and encryption. Encoding specifies how human-interpretable characters map to byte sequences. In the context of internationalisation, Unicode has emerged as the standard vocabulary with the eight-bit Unicode Transformation Format (UTF-8) being the dominant encoding. Still, a lot of data exists with legacy encodings such as Latin1 (ISO-8859-1). The compression packs similar data segments together to save space. For structured data, typically lossless compression is used, whereas for unstructured data (large corpora of text, images) also lossy compression schemes may be used for even higher savings. Examples of compressed file formats include ZIP (`.zip`), GZip (e.g. `.tar.gz`) and BZip2 (e.g. `.tar.bz2`), each related to custom compression algorithms and corresponding tools (packers, unpackers), with Tar being an intermediate format combining all files in one archive without compression. In data science, encoding and compression are major concerns whereas encryption is only used in specific circumstances, in particular due to the still emerging techniques to compute over encrypted data.

2.3 Compute-Centric Processing: Pipelines and Workflows

A pipeline is a sequential execution of instructions or programs. Pipelines can be represented as purely sequential DAGs, with edges referring to transitions between the individual instructions. The transitions therefore define a temporal and causal order between the instructions. The pipeline commences with input data received by the first instruction. The output of the instruction can, fully or partially, and optionally in conjunction with the original input, be forwarded as input to the next instruction. The output of the last instruction determines the result of the pipeline. Any instruction failure is either ignored or leads to the failure of the entire pipeline.

Fig. 2.2 visualises an exemplary pipeline. Especially for pipelines containing many instructions, the raised abstraction level and uniform interface with input, output and error messages makes pipelines a suitable structure and mechanism for data processing.

[2]COVID-19 schema definition: `https://data.tg.ch/api/datasets/1.0/dfs-ga-1/attachments/variablenbeschreibung_covid19_tg_pdf/`

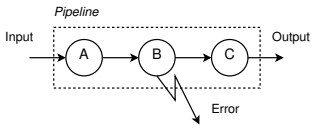

Figure 2.2: Pipeline structure for sequential processing

The instructions can be represented by tools, i.e. OS-level pipelines, or by microservices, i.e. pipelines as orchestrated service sequences. The handover of data (i.e. the edges) can happen as streams or through agreed-upon locations such as files. Moreover, when input and output interfaces do not fully match, the edges can be augmented with additional transformation and conversion nodes. Transformation typically refers to changes within one data format, whereas conversion may refer to changes in data format. Fig. 2.3 visualises an example of such an augmented pipeline.

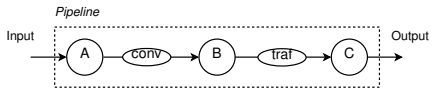

Figure 2.3: Augmented pipeline structure with transformation and conversion nodes

When sequential execution is no longer sufficient, branching and joining can be conducted. This leads to full workflows, essentially directed graphs with a single start-point and a single end-point. Each node of the graph represents one instruction. Depending on the result of each instruction, the branching can be controlled by either selectively, conditionally and exclusively entering one of the branches and leaving out the others or by parallel execution of all subsequent branches.

Textual notations exist for pipelines and workflows, although they are based on convention rather than standardisation and therefore often specific to the runtime environment. Two activities A and B may run sequentially without data handover, with or without activity failure causing pipeline failure, like this: `A && B` and `A; B`. Further variants are execution with data handover (`A | B` or `A >> B`) and execution in parallel (`A && B` or `[A, B]`).

Given that definition, pipelines represent a reduced form of workflows, and hence runtime support for both pipelines and workflows is broadly available from many workflow management systems, whereas basic pipeline support is also baked into operating systems. Each pipeline or workflow can be executed multiple times, with each instance indicating progress by a reference to the current activity.

2.4 Data-Centric Processing: Sharding and Map-Reduce

Pipelines and workflows emphasise the compute activities as first-class citizens, with secondary specification of what happens to the data. In contrast, other processing paradigms put data first and then specify how application logic is applied to it.

In this context, sharding and partitioning are terms that refer to splitting a large amount of homogeneous data records, for instance, lines in a text file or objects serialised as JSON, in a way that the resulting shards or partitions are equally sized or otherwise balanced. In case each shard can then be processed independently, for example, in a counting operation, the processing can be trivially parallelised. This speeds up the computation significantly, either to the number of available cores on a machine or in the case of distributed parallelisation even to all cores across a number of attached machines.

Sometimes, complete parallelisation is not possible, for example, in the operation to find the maximum of a large list of numbers. In that case, the algorithm is adjusted to parallelise as much as possible – for instance, finding the local maximum in each shard – and then in a second step to find the global maximum by working across the intermediate results from the first step.

The map-reduce programming paradigm combines both steps. First, functions are applied (mapped) to individual data records within a shard, yielding equally sized result shards, except for filter functions, which can also yield smaller result shards. Then, the resulting shards are combined in a reduce step. The following example shows map-reduce processing over a list of text lines in Python, by first converting all text to uppercase, and then counting and summing the occurrences of the letter A. It is apparent that there are no instructions to parallelise. Whether or not each map activity runs in parallel is decided by the implementation, whereas the programming can focus on working with the data.

```
import functools
text = ["Heavy snow", ...]
result = map(str.upper, text)
result = map(lambda line: line.count("A"), result)
result = functools.reduce(lambda x, y: x + y, result)
print(result)
```

2.5 Event Processing, Handlers and Hooks

There are multiple ways to start the execution of an instruction or a complex program. The first one is the conscious interactive invocation, for instance, by

the data scientist during development or by the user in production. The second way is programmatic invocation. An application may call another application to delegate processing tasks. The third way is time-triggered execution. A specific time pattern such as 'at the full hour' or 'every 10 minutes' is given and translated into action by a time trigger system such as an implementation of Cron[3]. Time triggers are predictable and can be anticipated. The fourth way is an unpredictable event trigger. A program is to be executed whenever something happens, such as changes in a data structure. The advantage is that, if the event rarely occurs, there is no unnecessary invocation and therefore no overhead cost. Within the processing logic, it is often less relevant what exactly was the trigger compared to what context information is available, i.e. how the trigger was parameterised.

Handlers and hooks provide the glue between source events and program invocation. For instance, data modification can be monitored over a long time, and whenever a modification is detected by a helper program, a specified program is invoked. This also works over the network, where appropriate network messages can be sent upon detection of source events, in the form of network hooks or web hooks. Hence, these handlers perform a similar role as transformers and converters in workflows.

The term handler is furthermore used for handling internal events in a program. This may be an exception, an interruption or some other signal from the environment. Operating systems provide handler support on an OS programming level, whereas workflow management systems also translate these to higher-level handlers to specify what should happen to a pipeline or workflow in the event of a fault or signal.

2.6 Encapsulation: Functions, Tools, Containers and Services

Encapsulation refers to the separation of the interface from the implementation. It is an important concept to hide implementation complexity by raising the abstraction level. Pipelines and workflows have already been introduced as abstraction layers and encapsulation mechanisms. This section takes a broader look at encapsulation.

Within a program, application logic might be encapsulated as a function with a well-defined signature, consisting of the function name, its mandatory and optional parameters, and its return values. Beyond functions, logic can be encapsulated as a class with multiple functions (methods) or as a module or library potentially encompassing multiple classes. The limitation to all of

[3]Cron syntax: `http://www.quartz-scheduler.org/documentation/quartz-2.3.0/tutorials/crontrigger.html`

these encapsulations is that they are only accessible within the application, demanding all programming be done with the same programming language.

It is possible to go beyond this limitation. The program itself then represents another form of encapsulation with the ability to parameterise the execution, capture intermediate information about its activity and eventually determine the success of the execution based on mostly self-declared information and conventions. The level of encapsulation is stronger if idempotency is guaranteed, such that multiple repeated invocations do not produce results different from a single invocation. This is typically achieved through statelessness, a property helpful also for the automated parallelisation. For data-centric programs, statelessness results from read-only operations that do not modify data, for instance, searching, whereas modifications lead to statefulness and then only rarely combine with idempotency. For instance, updating an entry to the current invocation time is not an idempotent operation, whereas nulling a fixed field is.

Smaller programs with well-defined input and output interfaces are conventionally called *tools* on the OS level. They might read data from files and over the network, as well as on the command line and through interactive input, and they might return results via files, network and standard output, in addition to an execution status code or exit code. Their invocation context encompasses explicitly given parameters in addition to configuration taken from configuration files and environment variables. More complex application functionality can be encapsulated as containers, with similar parameterisation.

Another form of encapsulation is network invocation of these programs via a well-defined service interface, which abstracts from implementation details such as the programming language. Such services have the advantage of being deployable and remote-invokable across machines. Often, the services take the form of microservices with specific forms of packaging and lifecycle management. Either they run continuously and directly accept requests, or a proxy takes requests on their behalf and invokes them only on demand. This latter form has become popular under the name cloud functions, conveying the conceptual similarity to programming-in-the-small function encapsulation.

2.7 Data Management, Engineering and Operations

Concepts such as data formats, workflows and event processing are generic and not necessarily bound to the activities of a data scientist. In this section, these concepts will be connected and explained from a data-centric angle. They encompass data management, engineering and integration, machine learning, operations (coined DataOps) and reproducibility.

2.7.1 Data Engineering

Data processing requires two main ingredients – input data and software that performs the processing. Building software in the form of applications and services, but also curating the input data, often starts with a repository managed by a version control system (VCS) to manage all ingredients and to track their evolution. Additionally, data may be managed in a relational or non-relational database (DB), in storage services, or in a data warehouse. To reduce the effort, when maintaining both modest amounts of schemaless or schema-flexible data and software code, a VCS with file-based data representation is a pragmatic choice.

A notable feature of VCS and DBs are hooks triggered whenever the repository or database contents change. For instance, whenever a VCS-managed file is modified, a user-defined script is executed that validates the repository's data files or submits the updated files to some online service. Similarly, User-Defined Functions (UDFs) may execute within a DB upon the insertion of records. Making use of such triggers for further processing leads to continuous integration processes, with reference to data integration explained in the next paragraph, or continuous deployment of data on a provisioning system, collectively referred to as CI/CD.

The processing itself can then take various forms. Apart from integration or fusion, data might be aggregated, augmented, analysed, filtered or used as basis for decision-making. In recent years, machine learning (ML) has gained significance, with large volumes of complete data used to train a mathematical model used for inferring characteristics of incomplete data records. For instance, a regression test on given X/Y coordinates may infer the associated Z coordinate by interpolating from known Z values of nearby X/Y pairs. Many ML algorithms work on vectors or tensors and benefit from specialised hardware to support concurrent vector processing. ML algorithms can be used to predict such missing values, but also to categorise objects described by data, and to recombine information, always based on statistics and heuristics with an imperfect accuracy. Recombination works on large language and media models (LLM, LMM) and is able to produce text and multimedia content with defined characteristics, although often not perfect due to the mentioned heuristics. Especially when generating results for human consumption, these techniques are also referred to as Artificial Intelligence (AI).

Moreover, data engineering is concerned with the right design of data-processing software, in particular with its non-functional runtime characteristics such as scalability, performance, reliability and cost. These characteristics take effect when the software is operated, especially under the DataOps paradigm. The design also involves the decomposition of software functionality into internal structures such as microservices and workflows and the preparation of automated deployment of the corresponding management systems.

2.7.2 Data Integration

Data integration refers to the ability to load and interpret data from any source and in any format. It requires a unification of formats through transformation and conversion. Transformation modifies structured data within one homogeneous format, for instance, by adding, removing or renaming fields. Conversion refers to the change between heterogeneous formats, for instance, from JSON to XML. Assuming there are N source formats and M target formats, this would require the implementation of $N \times M$ converters. Instead, the conversion can be conducted to and from a meta-format at the expense of slightly slower processing due to two conversion steps. The benefit of such an approach is that at a maximum, $N + M$ converters need to be implemented and maintained.

2.7.3 DataOps

When one combines processing paradigms with input data in various formats and activities in various encapsulations, including storage, transmissions and triggers across machine boundaries, the question arises whether this sort of programming in the large can be summarised under one term. There are several contenders, but, for pragmatic reasons, the term DataOps is used here to conclude the first concept chapter.

DataOps is not a rigorously well-defined term. Rather, based on the industry-invented term DevOps and related to GitOps and MLOps, it describes a rather wide set of activites around providing code and data as managed services to other users or applications, primarily based on network interfaces and microservice encapsulations, based on CI/CD. This implies attending to repository design, powerful data processing pipelines development and deployment, integration and inference, sufficient resource allocation, fair scheduling, protection, monitoring, cleanups and troubleshooting to provide flawless services. From the data science engineer, it requires a fundamental understanding of concepts and tools at the operating system level but also concerning data engineering and data science platforms and distributed infrastructure. The engineer needs to understand the contribution of any technology to the resulting data product, including business-affecting terms such as technical debt and dependency risks. Moreover, the engineer needs to be able to resolve emergent, sporadically occurring unexpected problems, apart from investing time and effort into longer-term improvements that may be requested by users of a data-centric application. Hence, from a data product engineering perspective, it may be more cost-effective and sustainable in the long term to build on proven portable technology stacks rather than the latest offerings, which may not exist anymore or become less economical within few years time. For the data scientist to be productive in the operations domain, there are also

certain requirements on powerful zero-configuration management tools, dashboards and actionable advice when problems occur. These requirements are only partially fulfilled by today's software, even when taking the latest data warehouse or data lake offerings into account, leading to even more emphasis on being able to master basic tools and foundational processes in order to build custom DataOps solutions in any business context.

2.7.4 Reproducibility

Data management requires a thorough documentation of technical and legal aspects concerning the origin of data, any modifications performed and long-term archiving. These processes are often subject to regulatory constraints, for instance, related to data protection. Yet having good documentation, especially an automatically generated one, also helps identifying improvements and regressions in data quality and data-driven implementation and process quality. A number of terms relate to the ability to reproduce results from the same input data. The goal is always 100% reproducibility, implying that the results do not change when there is also no change in the input data. The following four terms are specifically of relevance in this context:

1. Lineage. Data lineage is the process of maintaining the journey of data from its originating sources to its ultimate destination. It refers to the automation and exact documentation of all transformations and conversions on that journey.

2. Provenance. Data provenance is the historical tracing of data from its originating source to its final stage. Emphasis is on the source, i.e. is the data source credible, legal, of sufficient quality and so forth.

3. Reproducibility. In general, this represents the degree of agreement of a measurement by different individuals, locations and instruments. Specifically related to data science, it specifies the equality of results of data processing and analytics no matter what code or environment is used.

4. Repeatability. This denotes the variability of a measurement or of a pipeline or workflow across iterations under the same conditions. Typically, measurements are conducted multiple times and evaluated statistically. If the arithmetic mean average, median and spread are not far apart, the data processing is repeatable due to producing comparable results in repetitive iterations.

In data science practice, all four terms are important, and all require specialised tools and platforms to assist the data scientist.

Repetition

All repetition solutions can be found at the end of the book.

1. What relations may exist between two tasks of a workflow?
2. Why is the map function a suitable primitive for highly scalable applications?
3. A VC-funded startup offers analytics as a service. Should one consider a subscription?

Chapter 3

Concepts: Operating Systems

This section conveys background knowledge on the inner workings of a computer system, with emphasis on contemporary operating systems. The intention is not to give a formal education on all aspects, but rather to provide a practical-oriented jumping board to understanding any later interaction with the system. Most topics are touched on only briefly and require additional literature for full understanding. The chosen method is to touch on topics of further necessity in this book, and to give a technological foundation to the programming concepts presented beforehand. This section presents fundamentals including the boot process, current operating systems, technical building blocks such as process and resource management, isolation concepts, file system interaction and networking, and user management.

3.1 Fundamentals

An operating system (OS) is a complex piece of software that runs all the time on a given hardware to manage that hardware and facilitate the concurrent execution of multiple further software applications. Hardware can relate to interactive work devices such as notebooks, personal computers and mobile phones, but also to servers, as well as embedded applicances (such as programmable machines) and virtualised hardware. On a typical data scientist workstation, the core computer hardware could, for instance, be a set of n processors (central processing unit – CPU), an accelerator (graphics processing unit – GPU), x GiB of main memory, y GiB of storage on a local SSD, a wireless network adapter and a 10G wired network connection, along with a mainboard to tie these components together and peripherals for human input and output. The peripherals may encompass a monitor, keyboard and mouse, printer and scanner as well as loudspeakers. In an application scenario, a hypothetic dataset

might be first read from the SSD. The application software processes it according to program logic as a set of instructions similarly read from the same SSD, and the results are written to main memory. The OS must coordinate this workflow including permissions and auxiliary program execution. Hence, the OS performs tasks such as hardware initialisation, coordinated access to hardware resources, memory and file management, scheduling of tasks that are part of such workflows, permission checks and usage accounting, among many others. In terms of complexity, operating systems of practical relevance encompass millions of lines of source code.

Operating systems are activated by a *boot process*, taking over from the hardware's hard-wired internal initialisation routines such as Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) that are provided with the mainboard. They then activate the peripheral hardware, load initialisation code from the storage media such as disks, start system-wide background processes and prepare the system for human and automated usage. The usage is facilitated by offering local and networked interfaces to log in and run commands, primarily in the form of textual or graphical login screens or remote login services, again either textual or graphical. Nowadays, the boot process in both physical and virtual machines is often complicated by the presence of mandatory Trusted Computing Modules (TPMs).

The hardware initialisation depends on the degree of standardisation. Some hardware models are easy to integrate from the OS perspective, whereas most others ship with their own firmware requirements that need the firmware to be uploaded to the hardware first. This is often the case with wireless networking adapters, without which a network-based installation or maintenance becomes impossible. Hardware unable to work without firmware updates is another cause of complicated boot processes. Modern operating systems cover a broad range of hardware but may lack firmware or driver support for exotic models or very new models.

The entire boot process usually takes a few seconds. To avoid costly reboots, hardware can also be suspended or put into standby when it is not in use, for instance, by instructing the OS to do so or by provoking that with predefined actions such as closing the lid of a notebook. Correspondingly, hardware can be woken up again either by user interaction (opening the notebook lid, pressing a key, touching the screen) or by a network signal (wake-on-LAN). However, most system suspensions still consume a certain amount of standby power, which has not only the implication that more electricity needs to be generated, but also that the suspension does not survive a longer power cut.

On a new computer without pre-installed OS on permanent storage, apart from netboot in a prepared environment, the OS can only be loaded from removable boot media such as USB drives or CD/DVD, which would then provide an installation option. Alternatively, in case an OS is already installed, some

systems provide installers running as applications on other systems, which allows for complementing or replacing an existing OS. For that matter, the disks of a computer can be partitioned. An OS requires at least one such partition for booting, containing the boot loader initialisation code in the first sector. An OS can, however, also span across multiple partitions to increase the storage capacity.

3.2 Current Operating Systems

The scope of operating systems has changed considerably over the decades. Today, despite an unprecedented variety of available operating systems (to get an idea, occasionally read up on OS-related news websites[1]), only very few have practical and commercial relevance in the market and satisfactory support for applications, especially those that integrate, process and visualise data. Some systems like Minix are suitable only from a didactic perspective to understand the inner structures without broadly supporting today's hardware and applications ecosystem. Others like the Robotic Operating System (ROS) refer to industry-specific middleware on top of an existing OS kernel.

The three dominating contemporary operating systems for data science interaction and DataOps, both vendor-neutral and vendor-specific offerings, are those that are used on workstations. They thus define the immediate environment for the data scientist, allowing for rapid interaction between programming in the large and verifying the results of program execution:

1. Linux[2], with several different representations including flavours of GNU/Linux (e.g. Debian, Ubuntu, CentOS), and similar open systems with similar userland such as various Berkeley Software Distribution (BSD) flavours;
2. Apple Mac OS X, similar to Linux, being of the Unix family of systems; and
3. Microsoft Windows.

From a consumer perspective, these operating systems are often known from mobile devices such as the Linux-derived Android or the OS X-derived iOS. However, apart from mobile data acquisition and visualisation, such devices, including mobile phones and smart watches, do not play a major role in defining infrastructure for application programming and data processing.

Linux, the BSDs and Mac OS X both follow a similar technical interface design as the original Unix systems, and are therefore also referred to as Unix-like systems. Unix systems date back to the 1970s, and some elements of that

[1] OSNews: `https://www.osnews.com/`
[2] Linux Kernel: `https://kernel.org/`

era are still reflected in modern systems, including conventions on timestamps and programming languages. Each operating system is nevertheless characterised not only by the actual up-to-date OS kernel but also by the supported subsystems such as file systems, device drivers, input modalities and finally the supported and still growing ecosystem in terms of libraries and applications. The collaboration models facilitated by the Internet have led to large global communities contributing to the software development from the OS level to the applications. As the ecosystem is therefore neither static nor centrally coordinated, new software is constantly being published, and existing software might need to be updated to fix stability and security issues. Consequently, concepts of trust and reputation regarding software found online are important. To reduce the setup effort, curated distributions, package managers, software stores and similar consumer interfaces to the ecosystem, along with corresponding producer interfaces, exist and are central for an effective usage for setting up data science infrastructures. There are OS-specific and cross-OS (or cross-platform) applications and libraries within each ecosystem, with further differences in the hardware architecture. Fig. 3.1 conveys typical software-architectural layers found in current operating systems, running within a privileged OS scope and an unprivileged scope defined by the respective OS distribution and managed by the OS itself at runtime.

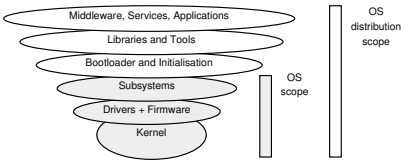

Figure 3.1: Typical layered system architecture

For reasons of practicality related to condensed explanation of commands, and reflection of the high pace of the ecosystem, although the operating systems theory are presented in abstract form, concrete usage examples in most cases assume a running Linux system. Students running different operating systems are either able to relate the commands to equivalents on their systems or use virtualisation or network access to get their hands on a native Linux environment.

3.3 Building Blocks: Executables, Processes and Resource Management

Most users know computers in terms of applications. For instance, a data scientist may create a Jupyter notebook (detailed later) and use the Jupyter

application for that purpose. The visible part is the notebook shown on the screen, with cells to input code and other cells to display results; but of interest from an OS perspective is the underlying structure. They determine how the Jupyter notebook got found, started and delivered to the user.

Applications within the ecosystem of an operating system are shipped in the form of bundles or *packages* consisting of files and, sometimes, online services. The central parts of any application are called *executables*, referring to single files that can be executed on a CPU under the control of the operating system. They contain processor-specific binary code in an umbrella file format understood by the OS, such as *ELF* or *EXE*, whose headers give information on how to load and execute the file. The production of such executables is the task of a compiler that takes a human-readable language, either Assembler or a higher-level language such as C, C++ or Rust, and optimises the set of instructions for the processor.

Executables can be statically or dynamically linked to essential library code. Such *libraries* extend the functionality of executables with re-usable program logic, including functions to perform basic interaction with the operating system. Based on the historic dominance of the C programming language in operating systems, the main library is the standard C library, or `libc`.

In the case of dynamic linking, especially for highly dynamic *plugins*, the code is resolved upon execution by the OS *loader*. The advantage of dynamic linking is that the code size of the executables themselves can be kept modest, at the disadvantage of increased complexity during loading. In practice, the executable does not run unless all library dependencies are properly resolved, including the determined versions.

If the application is authored in a scripting language, such as Python, then, from an OS perspective the application itself is just data, and the Python interpreter is the actual executable. To make matters complicated, there are multiple Python interpreters available, such as the classic cPython, iPython and PyPy. Multiple applications can also run at the same time with coupling between them through communication. The Jupyter notebook environment would be one such case, which is itself developed in Python, typically executing on cPython and running an iPython interpreter instance to execute the content of its cells.

Before an executable can run, the OS also needs to prepare its internal structures. It allocates a new *process* that refers to the binary code in the executable but also contains management information such as process identifier, parent process identifier, current working directory, priority, owner, access privileges, resource limits and statistical information about the ongoing execution. In practice, a typical data scientist device runs around 300–500 processes concurrently, many of them referring to invisible background activities. To give a more reasonable example, the aforementioned execution of Python code in a

Jupyter notebook is explained in Fig. 3.2. It highlights the uniquely separated aspects of processes such as identifiers, owners and memory areas.

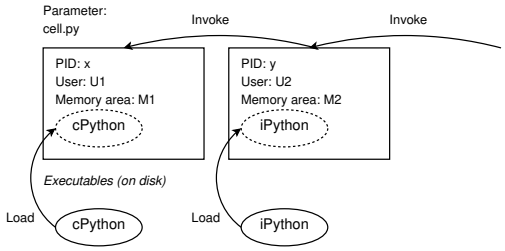

Figure 3.2: Processes, executables and data in an operating system context

To run correctly, a process needs a certain amount of resources. This refers primarily to CPU and main memory, but depending on the program logic may also refer to disk space and network throughput.

By default, processes can use all virtual memory made available by the operating system. This value often exceeds the available physical memory due to concurrent process execution. Hence, processes may enter into situations where no more main memory is available. When other forms of memory (e.g. swap space) are also exhausted, the OS decides on the termination of the offending process or another process depending on the memory management policy. Eventually, one process must be terminated with an out-of-memory (OOM) cause. Then, more memory becomes available again, although not necessarily in sufficient quantity to fully accomodate the original request. Unless it runs into such critical situations of resource exhaustion, each process nevertheless requires different amounts of memory over time. The amount is hard to predict, although often similar if the process is run with comparable parameters. One task of the operating system is to shuffle with memory pages behind the scenes, reallocating them as needed to the active processes.

Processes can also terminate irregularly for reasons other than out-of-memory conditions. The primary reason may be an internal programming mistake such as division by zero or access to a file that does not exist. Programming languages allow for capturing such mistakes as exceptions, but if that was forgotten, the process is terminated by the operating system.

Processes run with configurable priorities. If a batch process, such as data cleansing, runs quietly in the background and does not disturb interactive work, it can be explicitly de-prioritised. On the other end of the spectrum are processes that require a minimum allocation of CPU slices, for instance, to be able to perform guaranteed real-time processing of a large and fast data stream. Most operating systems have no support for hard real time processing and instead work on a best-effort basis.

At any point in time, an OS thus runs a number of processes, from which more processes may be spawned. Each process possesses a priority and an associated memory allocation apart from the other mentioned characteristics. All processes form a tree, with the initialisation process as its root. This process is typically instantiated from a special application program that gets a list of other processes to start at system boot and supervises them to be able to conduct restart in the case of crashes. On modern Linux systems, this task is performed by OpenRC or SystemD.

3.4 Isolation, Virtualisation and Containerisation

As outlined in the programming concepts, encapsulation is an important principle to raise the abstraction level and shift from programming in the small to programming in the large. In operating systems, the main encapsulation techniques relate to process-level isolation, virtualisation and containerisation.

Processes that run concurrently have a unique identifier within a single namespace and are protected from mutual access to main memory. Only the owner of a process, or a privileged superuser can change process metadata such as its priority. The system may also enforce a certain fairness between users by setting quotas that effect the number of processes that can run, the amoung of main memory allocatable by each process, the number of open files per process, the cumulative CPU time and similar metrics. However, in principle, the separated processes can see each other and might volitionally or inadvertantly share information through the file system or other less protected channels, including non-intuitive covert channels. In order to prevent information sharing and interference, specifically for shared user environments, different *isolation* concepts beyond regular OS process isolation have emerged.

A strong level of isolation can be achieved by *virtualisation*, i.e. running an application that represents a guest operating system including kernel and is thus able to execute applications within itself, without visibility of its internals to the underlying host operating system. The guest OS can be a different version of the host OS or even an entirely different OS, for instance, when running Windows atop Linux. In principle, virtualisation works even without specific support in the host OS but is then often too slow. Modern OSes thus support concepts such as para-virtualisation and hypervisors that provide near-native execution speed in conjunction with specific CPU instructions. Depending on the device configuration, virtualisation can even be nested, so that complex applications can be built in a portable way. The management of guest OSes is the responsibility of a *hypervisor*, which is usually built into host OSes nowadays and apart from the runtime functionality offers ways to

configure virtual machines (VMs) as well as manage VM images that contain the root filesystem of the guest OS. Typical examples for native hypervisors are KVM on Linux and HyperV on Windows. There are also third-party hypervisors such as VirtualBox for all maintream operating systems.

A weaker but in practice often sufficient level of isolation is provided by *containerisation*. In this model, the OS kernel is fully shared and the process namespaces are decoupled selectively. For instance, a guest process may still share the networking with the host but not the process table or the file system. In contrast to most virtualisation approaches, containerisation is more optimised for software development environments and thus also suitable for running most software of relevance to data scientists. The first widely used common cross-platform approach for containerisation was *Docker*, providing management capabilities for container images and runtime aspects. Tools like `Podman` make it easy to work with such container images.

Virtualisation and containerisation can be combined with networking (explained below) to achieve flexible distributed computing. In this model, each operating environment should indicate its name, as otherwise the concurrent work across several systems quickly becomes confusing for the user.

3.5 File System, Paths and File Access

Files are sequences of bytes – either representing human readable characters (text files, e.g. Python scripts or structured data) or arbitrary bytes (binary files, e.g. executables or unstructured multimedia data). The native representation of data could be modified in a file context, as outlined in the explanation about data formats. For instance, text data might be compressed to use less space and encrypted to be more confidential, resulting most likely in a binary-encoded file.

For permanent access, such files are stored on storage devices (e.g. SSD) and are written and read sequentially in both text and binary mode or with random position access in binary mode. Alternatively, they are memory-mapped for direct access, especially larger data files for which a sequential access might be too much of a bottleneck. The task of the OS, more specifically of the OS-supported file systems on any storage media, is then to organise all files, arranging them, allocating the physical storage locations, and regulating as well as journalising access.

File systems therefore arrange files in suitable structures, most of which are hierarchical or tree-structured. They emerged as integral parts of operating systems, and most optimised file systems remain OS-specific (e.g. ZFS, ext4), although many de-facto standards such as the exFAT file systems have emerged along with portable media such as USB drives. In general, file systems require a (physical) storage medium such as a hard disk, solid-state drive

or the mentioned USB drives; or alternatively, a partition on that medium. Multiple partitions can be integrated as additional file systems located as subdirectories or drives depending on the OS type. At least one partition is marked as bootable and contains the OS files along with the bootloader, reachable from the medium's master boot record. Being the central interface between humans and data management on the OS level, file systems are designed to balance user needs in terms of structuring data with operational concerns such as file creation and search speed as well as resilience.

The user perspective on a file system typically encompasses a directory tree view. Directories with arbitrarily nested subdirectories form tree structures that, along with file contents and metadata, assist users in data management. The top-most directory is called root directory (i.e. `/`) or drive (e.g. `C:\`), depending on the OS. The concatenation of directories, and optionally a file, is called a *path*. Paths can be absolute, starting from the top-most directory, or relative to the current working directory of a process. For instance, `/etc/passwd` is an absolute file path, and `..` is a relative directory path, referring to the root directory `/` when applied to the directory `/etc`. In addition to data and executable files as well as subdirectories, and depending on the file system and operating system, directories may also contain symbolic links (symlinks), device files, fifos and other special files.

Standard top-level directories on Unix-like systems encompass three hierarchies: `/`, `/usr` and `/usr/local`, with the semantics that larger partitions could be mounted later in the boot process and only `/` must unconditionally exist for booting and system repair activities. Under each of these hierarchies, certain second-level directories may exist. These include `bin` and `sbin` for unprivileged and privileged binaries, and `lib` for shared libraries and other architecture-dependent files. Further directories only exist in the top-level hierarchy, including `etc` for configuration, `home` for user accounts, `tmp` for temporary files, `opt` for optionally installed large applications, `root` for the super-user account, `mnt` for mounted filesystems, `srv` for services and `var` for generated state information. The `share` directory for architecture-independent files, effectively all data files read by applications, only exists within the `/usr` and `/usr/local` hierarchies, presumably because the core utilities under the root hierarchy work as executables without references to data files. These directories are further subdivided into purpose and application Hence, a typical application would consume data from `/usr/share/<app>`, write temporary data into `/tmp/<app>[/<user>]`, cache results into `/var/cache/<app>`, and persist precious data into `/var/lib/<app>`. Some directories contain OS-related support files and special files (`boot`, `dev`, `proc`, `sys`). These names and conventions have evolved over time and are now largely standardised due to the Filesystem Hierarchy Standard (FHS).

As outlined above, files are characterised by their contents, either textual or binary, as well as their metadata. While content is always defined by the

user or applications on the user's behalf, some of the metadata is automatically maintained by the file system. Typical text formats encompass unstructured plain text in various encodings, but primarily Unicode (i.e. UTF-8) as well as structured plain text to represent data such as CSV, XML, YAML and JSON formats. Such files are interpreted as sequence of lines, where each line is separated from another by an end-of-line symbol, typically either the newline symbol (`\n`) and/or the character return symbol (`\r`). Binary files, on the other hand, have application-specific structures that render them unfit for human reading. Many data science tasks require working with structured data formats and understanding their content. This implies dealing with potential quality issues, including misformatted files that would disturb the process of reading or *parsing* but also silent errors that would cause issues after the parsing.

Files are also characterised by their metadata. These include auditable timestamps of creation, modification and last access but also size, ownership and access permissions. The file size may not necessarily correspond to their content, for sparse files may be created conveying a large file size despite little content and correspondingly small storage space requirements.

From a programming perspective, files are opened in a certain access mode, typically for reading, (over)writing or appending. The OS checks the eligibility of the file open request and either signals success by returning a file descriptor or signals an error through appropriate OS-specific error codes. Typical errors include 'file not found' and 'permission denied'. Subsequent read and write operations may similarly result in errors such as 'no space left on device' or – specifically for reading – in application-generated parser errors. At the end of a file operation, the file is closed again and the OS-internal control structures are released.

A trivial way to read files is to read them line by line (for text files) or block-wise (for binary files). In order to cope with very large files and to avoid the input/output bottleneck, modern OSes allow the virtual mapping of files into memory, so-called mem-mapped files, on the assumption of sufficient main memory. Any modification in main memory then results in an optimised write access on the storage medium.

3.6 Networking

Network access across the boundaries of single computers is useful for a number of reasons. It allows fetching data from all over the world, along with code in the context of system maintenance, to backup data remotely and to let users collaborate on data science projects. In conjunction with file systems, networking also allows for remote file management through networked file systems.

On a practical level, networking is considered to consist of four layers. On the lowest level, a physical link between two computers is established using either wired or wireless communication media, such that a computer becomes reachable from another one by a network address. This link may be direct, but it might also involve a number of intermediate machines, physically represented as a sequence of links. The physical link determines the key networking characteristics: throughput as in data volume per time unit, latency as in time unit that needs to pass before an arbitrarily small message has reached the destination and stability, which in the case of wireless links is affected by the signal quality among other factors.

On the next level atop the physical link, a packet communication is established, dominated by the Internet Protocol (IP) for data transfers as well as several network management protocols. Each IP packet consists of a header and a payload of fixed maximum size, typically around 1500 bytes but in principle up to 64 kiB. The IP abstracts the underlying physical linkage away so that the endpoint computers would always assume a single logical link in between them. Endpoints are specified as a combination of IP address (IPv4 or IPv6) and a 16-bit port number, with all numbers below 1024 being in the reserved range. Services can occupy one or multiple of those ports, and clients specify them when attempting to communicate with a service. The `/etc/services` file on Unix-like systems or `C:\Windows\System32\drivers\etc\services` on Windows maps relevant port numbers to human-interpretable names, in most cases protocol acronyms, as defined by the Internet Assigned Numbers Authority (IANA).

On top of IP, encapsulated in its payload section, data transfer with session or stream semantics across several individual packets is the task of more specialised protocols such as the User Datagram Protocol (UDP) for low-latency transmission, Transmission Control Protocol (TCP) for reliable transfers and Stream Control Transmission Protocol (SCTP) with characteristics from both, each of which again consists of headers and payloads. To secure connections, especially TCP connections can be upgraded with Transport-Layer Security (TLS), allowing for mutual certificate checks to verify identities and set up encrypted transfers.

On the highest layer, application-specific protocols define the content of the payloads and the sequence of transmissions as well as the details of addressing. Many applications assume a client-server topology where a server listens for packets on a specific numeric port, often chosen from a fixed assignment such as port 80 for web applications using the Hypertext Transport Protocol (HTTP), but in many cases with flexibility especially in the non-reserved range of ports. A client then connects to that port, establishing on its side a sending port with a random port number. Either side may initiate the conversation and may terminate it. Similar to files, networking protocols might be text-based (e.g.

HTTP, SMTP, XMPP) or binary (e.g. SSH), and data representation on the network may be subject to modification such as compression and encryption.

In IP networks, the IP address is the native address, although memorisable alias names can be given. This includes the computer's own internal name, independent of the network configuration, as well as network-reachable names including the computer's own external name. Unless the configuration is local, all network-reachable names can be retrieved from the Domain Name System (DNS), which is one of the standard services supporting network administration and navigation. Local configuration can reside in the file `/etc/hosts` on Unix-like systems, mapping IP addresses to local hostnames per line, e.g. `192.168.0.5 mynas`. On Windows, this file is called `C:\Windows\System32\drivers\etc\hosts`.

From an OS perspective, *sockets* are used as internal memory structure to represent active network connections from clients to servers and listening connections on servers. Each port number may only be in use once per machine. Therefore in conjunction with virtualisation and containerisation, virtual networks are introduced to route requests, for instance, to port 80, to different physical ports of the respective virtual machines or containers.

Sockets together with files and named or unnamed (pipe) FIFOs permit narrow-band inter-process communication (IPC). For sharing access to large data volumes efficiently between local processes, shared memory areas may be defined, whereas for communication across nodes, sockets are the only option.

An application offering functionality over a network port is typically called a *service*. For managing and providing such services at scale, several *service platforms* and *cloud platforms* have emerged, with some also offered on a commercial basis. Similar to OSes, many different platforms are available. Global-scale providers such as Google Cloud Platform, Microsoft Azure, Amazon Web Services, IBM Cloud or Alibaba are among the well-known ones with the richest portfolio of platform functionality, although there are also many local providers offering at least basic services and service management over the network.

3.7 User Management, Authentication, Authorisation and Credentials

The concept of users, roles and identities in a complex, shared system environment, especially in a networked environment, is an important one. Once the identity of a user or application is known, permissions can be linked to it and codified into the file system and other OS structures. The key question is then who is allowed to access or modify which files under what circumstances. On the OS level, system users are registered along with a home location in the

filesystem and credentials, typically in the form of passwords, and appropriate privileges, typically in the form of being in a group that has wider access permissions. Specifically, super users or root users have all possible permissions, and users are able to obtain these permissions temporarily. Hence, each OS process has a user identification and a (short-term) effective user identification that governs the permitted actions such as access to files and devices. On a Unix-like system, the super user's home directory is `/root`, whereas all other users typically reside in a user-specific directory such as `/home/user`. On Windows, the equivalent path is `C:\Users\user`.

Obtaining the permissions is called *authentication*, whereas making use of the permissions for access purposes is called *authorisation*.

To enforce authorisation across computer boundaries over a network, *credentials* such as passwords, keys or tokens are registered in services and supplied from clients during protocol-specific remote authentication. For increased confidentiality and to protect against leaks, the credentials are often not stored in plain but instead in hashed format. Moreover, they are also encrypted during transmission, based on timestamps as part of a session-specific encryption.

Repetition

1. What happens to a process when it requests additional memory pages but all main memory has already been used up?
2. Does containerisation provide the maximum possible isolation level between processes?
3. What happens when a web browser is used to connect to the website at `http://my-little-website`?

Chapter 4

Concepts: Infrastructure

The section informs about important concepts and goals related to digital infrastructure in the wider senses of data science. Therefore, it covers computing infrastructure, data infrastructure and similar forms, all of which are built atop operating systems and made available in various appearances such as compute clusters, online services and integrated platforms.

Similar to civil infrastructure such as roads and bridges or energy infrastructure such as power lines and gas tanks, digital infrastructure is a necessity with high demands on reliability, security, performance, scalability and cost effectiveness. Digital infrastructure relates to computing resources for computation, communication and storage. Sometimes, the handling of these resources is tightly combined. For the purpose of introducing typical infrastructures, the following types are covered here: networks and Internet, networked computers, services and platforms, high-performance computing and cloud computing.

4.1 Networks and Internet

Access to computer networks was already explained in the networking section of the operating systems concepts chapter from the perspective of a single system. Here, a bird's view on entire networks is given. Computer networks can be represented as graphs, with each system being a node and the available connections being edges. Not all edges need to be available or activated at all times, and sometimes more than one connection (e.g. wired and wireless) exists between nodes, making the nodes multi-homed due to being in different networks with different IP addresses at the same time. Hence, even the basic topology of a computer network is dynamic. It is also heterogeneous due to different characteristics of the connections related to bandwidth, latency and connection quality. Wired connections typically show predictable bandwidth

despite relying on slightly heuristic protocols, whereas wireless connections vary wildly. More dynamic behaviour is introduced by traffic flows, caused by the communication between nodes. Such traffic can lead to congestions on the network or overload of the attached systems. Unreliable connections can also lead to a loss of data. Overall, computer networks must be assumed to be dynamic, heterogeneous and imperfect. Fig. 4.1 summarises these characteristics in a topology in which nodes can have different roles at the same time due to being connected to multiple other nodes and applications concurrently. A node may serve as peer for video storage and as client for obtaining weather data.

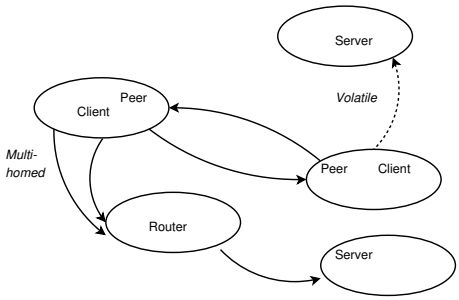

Figure 4.1: Exemplary computer networks topology and traffic flows

Computer networks may exist within physical premises as home network, corporate network or intranets, collectively called private networks. Access to services on these private networks from outside is then often physically impossible, adding an additional layer of protection on top of packet filters and application-level firewalls. The nodes within these networks have private IP addresses that are not routed on the public Internet. Any computer without such protection and with a publicly reachable IP address participates in the Internet. Virtual private networks can be used to extend the protection of private networks across the whole Internet.

IP addresses are either of type IPv4 or IPv6. In IPv4, addresses consist of four blocks of 8 bits, or 32 bits in total, in the form `a.b.c.d`, with each letter consisting of the values 0 to 255. Subnets are formed from right to left through a subnet mask. Hence, `a.b.c.0/8` refers to all addresses possible with d being 0 to 255, and `a.b.0.0/16` is a larger subnet with 64k addresses. In IPv6, there are eight blocks of 16 bits, or 128 bits in total. The global IPv4 address space is maintained by the Internet Assigned Numbers Authority (IANA), with Regional Internet Registries (RIRs) further allocating and assigning addresses to registered providers. The address space has become almost exhausted over recent years, in parallel with more widespread support for IPv6.

4.2 Networked Computers

A networked computer is a node on a network reachable by others on the same network or even globally, offering a certain number of services on one more multiple network interfaces. Three levels of access are typically distinguished: On the local interface (localhost), represented by the IPv4 address `127.0.0.1` and/or the IPv6 address `::1`; on a local network interface with locally addressable IP address such as `192.168.x.y`, `172.16.x.y` or `10.x.y.z`; and on a publicly routed IP address that, unless directly associated to a computer's interface, is network-translated by a router or gateway.

The quality of service on networked computers depends on the hardware resources available on that computer and on others attached to it as subordinates, i.e. slaves or workers, if applicable. Services may be offered primarily as storage and compute services. Two non-functional properties are of primary interest for compute services: performance and scalability. The performance indicates in which time an incoming service request or batch job is processed by the computer. Scalability is strongly related but indicates how many parallel requests or jobs can be processed without major regression in performance and without having to reject requests. For storage services, the performance (i.e. read and write access times) remains an important metric, but the capacity for storing data is also crucial, as is the integrity and further concerns. After all, computations can be restarted if temporarily lost (even at a cost), whereas loss of data may be irrevocable.

Attaching a computer to a network brings up security and trust issues. Especially on the Internet, access from other locations cannot be trusted per se, and appropriate security measures need to be taken. This refers primarily to keeping the system up to date (patched), running only the necessary services while shutting down others, introducing packet filtering and application-level firewalling to throttle requests and blacklist malicious origin sites, and performing offsite logging and intrusion detection to aid in post-mortem analysis.

From a data science workstation perspective, this means that complex services like those presented in the sections on middleware and platforms should be running only on localhost during the development phase without being exposed to access from outside, and that remote backups (potentially via version control) of at least all human-generated artefacts should be regularly conducted.

4.3 Services and Platforms

Services encapsulate application logic and further functionality behind a well-defined network interface. They are typically long-running processes exposing and expecting a higher-layer protocol on a specified port number. By connect-

ing to that port in a longer session or occasionally, an application becomes loosely distributed because some of the logic is executed in another process. Services in the wider sense encompass both programmatic and human interaction. Humans as service consumers interact with websites or virtual desktops among other human-computer interfaces, using specialised client applications such as web browsers or desktop widgets. Programmatic interaction on the other hand exposes Application Programming Interfaces (APIs) based on textual or binary network protocols. The interaction in terms of message contents and message exchange patterns is then controlled on the code level, either by directly composing the messages or by leveraging programming language-specific Software Development Kits (SDKs).

From a service consumption side, the service functionality is of primary interest. Services in the field of data science may offer the definition of projects or spaces, data storage and retrieval, data transformation and format conversion, queries, model serving, execution of predefined or supplied code and many other functions.

From the service provisioning side, notable configuration includes the network interface to bind to, the port number and permitted authentication methods, if any. As the service listens on the port number to handle incoming requests, it would have to interrupt the listening during the handling, leading to poor scalability. For this reason, multiple techniques exist to parallelise request handling. The first is the threading model, in which for each incoming request from a client, the service spawns a new thread that handles the communication with that client. This is very fast but also requires careful programming due to all threads sharing the same address space with a low degree of isolation. The second is the process model in which a full OS child process is created per request. This takes a bit longer (still in the milliseconds range though) but offers stronger isolation. The third is the proxy model, in which the actual processing is conducted on another machine and the service endpoint only performs input and output redirection, which saves CPU time. Various combinations and optimisations exist such as maintaining pools of prespawned threads or processes. Apart from high scalability, services can also be outfitted with high availability in terms of multiple instances. Requests may be redirected to one of those instances through load balancing at the client, DNS or proxy level, or they may be redirected to standby machine in case the primary machine is no longer available.

Services may be made discoverable with a well-defined declarative interface description or service description. Such descriptions first define permissable input and output messages including composite types on the interface level, but they can also cover messaging patterns, example requests and replies, and other aspects. Service descriptions with languages such as OpenAPI or the RESTful API Modelling Language (RAML) are common especially in the

field of web services that communicate over HTTP with pairs of requests and replies.

Service-oriented platforms encompass multiple complex multi-tenant software services that are typically accessed by users through an overarching web interface and connected to data processing services and backend services such as databases or file storage. Their architecture often follows a three-tier design with frontend, processing logic and persistence tiers, where each tier is not necessarily monolithic but can also be composed of multiple services, including microservice compositions. Platforms either derive users and permissions from the underlying operating system, or define their own model with the possibility to manage users, groups, roles, permissions, authentication methods and credentials. Many times, users can then generate credentials on their own, such as API keys, to facilitate the binding of clients to programmatic service interfaces.

Such platforms can be self-hosted or operated by a commercial entity. The former requires access to computing infrastructure as well as sufficient knowledge and skills of operating system administration. The latter types require registration for accessing most functionality. Both operational models may pose different risks from a privacy and security perspective. For proprietary commercial platforms, additional vendor lock-in risks exist.

Online services and platforms can be differentiated not only from a technical and operational perspective, but also on the business and legal terms of their operational model. Often, mixed models are used. A platform may require registration and filing a credit card, but gives a certain amount of free resources for first-time users before requiring the users to choose from regularly priced plans, either with fixed monthly price or with pay-per-use accounting and billing.

4.4 Parallel and High-Performance Computing

As explained for the request handling in services, scalability can be achieved through multi-threading or multi-processing, especially in conjunction with multi-core hardware. In general, multiple processors (CPUs/GPUs) can concurrently process partitioned data or perform other parallel activities. As a result to the denser use of computing resources, the overall processing time (wall-clock time) shrinks almost proportionally to the number of processors, although constrained by overheads. Those are not always predictable but are captured by a number of mathematical laws such as Amdahl's law, the law of Gustafson-Barsis, Sun-Ni and others leading to upper bounds for the possible speedups through parallelisation. Programming frameworks such as OpenMPI support the practical acceleration through parallelisation down to the code level, for instance, by detecting loops over data structures with no

causal dependencies between iterations. Message passing to synchronise parallel processes and tolerance for process failures are among the features that make parallel programming possible for average-skilled engineers. The support also happens at the data level with frameworks proposing suitable partitioning schemes so that the resource utilisation is maximised. From a cost and sustainability perspective, one should nevertheless be aware of the increased resource use. Slower batch processing may be an alternative to wall-clock time reduction for some scenarios.

Parallel computing with dedicated programming frameworks is sometimes offered as a distributed service for modest speed-ups beyond the number of CPU cores in a single system, i.e. for two- and low three-digits numbers of CPUs across systems. This evolves parallel computing into distributed parallel computing by offering capable high-level interfaces on top of the basic crunching. These interfaces allow for job submission and automatic distribution across machines along with the already mentioned functionalities known from parallel computing. Often, the distributed computing is then also subject to heterogeneity effects, with some CPUs finishing faster than others or some network links being slower than others, making predictable computing times a challenge.

High-Performance Computing (HPC) speeds up computation even more by massive parallelisation across a high number (hundreds or thousands) of compute nodes within a larger system or cluster, sometimes also within a supercomputer. Application programs are prepared for use on these clusters by internal parallelisation as well as primitives to synchronise the parallel activities such as independent loop iterations, often with message passing. The compute nodes are not freely accessible, and the compute time cannot be chosen freely. Instead, a compute job is defined, consisting of the program and necessary input data, and scheduled to be executed at the next possible time, within the allocation constraints. For instance, a user may be given a compute time of 1 hour across 20 nodes, and the program should be designed to maximise this allocation, ideally keeping partial results and being able to continue the computation in case of exceeding that allocation. A workload manager such as Slurm is taking care of the scheduling along with monitoring and production of execution statistics. Hence, HPC is suitable for batch processing of computationally intensive jobs such as number crunching and statistics, and often supported by specialised OSes such as Cray OS or Raise OS apart from tuned vanilla Linux setups.

HPC clusters are operated commercially but also within universities and national computing centres, covering scientific applications but also several data science-related tasks. In addition to the compute nodes, they often provide login nodes on which users can prepare the jobs and inspect the results. The global Top500 list informs about the peak performance achieved with HPC

machines every couple of months, but also in this community, sustainability concerns have become more important with the Green500 list, PUE and energy efficiency metrics and other indicators.

4.5 Cloud Computing

Cloud computing refers to the provisioning of applications, middleware and data through a set of infrastructure and platform services, collectively and colloquially called a cloud. These services are programmable and elastically scalable, they are provided on demand and, in a commercial context, they are metered and billed depending on the usage. The scope of the platform services relates to the platforms mentioned in the previous section. It encompasses the fully managed hosting of applications as virtual machines and containers, of data through various storage and database services along with other middleware, of network-specific functionality (DNS, HTTP gateways), of development services (Git, CI/CD), of large-scale data processing and machine learning interfaces including HPC and of many other functionalities.

Private cloud computing infrastructure is self-operated atop virtualisation and containerisation technology. Examples for these so-called cloud stacks are OpenStack, Kubernetes and OpenShift. For reliable storage, examples include Ceph and MinIO. Some of those stacks are also prepared to be run locally for development purposes, such as various flavours of Kubernetes including Minikube, Microk8s and K3s. Nevertheless, operating such basic infrastructure is typically outside the scope of a data scientist or engineer even with a faible for DataOps. A small misconfiguration can have serious and irreversible consequences such as data leak or data loss. On the positive side, a private cloud gives a maximum amount of freedom and flexibility, especially for trying out new technologies.

Many smaller commercial cloud providers exist on a national level in case a private cloud is not an option. Typically, these public providers operate their service portfolio based on existing cloud stacks. However, they do often not extend beyond simple application deployment and hosting, and their physical presence in many cases encompasses a single data centre or a pair of primary and secondary (failover) location. For most Small and Medium Enterprises (SMEs), this is more than sufficient, but a lack of support for diverse deployments (edge, serverless, accelerated computing) may become apparent soon even for them. A commercial but not-profit-oriented variant are institutional clouds operated, for instance, by national research and education networks for academic purposes.

Large multinational cloud providers, the so-called hyperscalers, all provide fully managed services for all target groups including a rich variety of offers for data scientists. Often these are provider-specific and sometimes even domain-

specific, such as analytics services for the health domain. These providers operate dozens of data centres in multiple regions. Additionally, the hyperscalers operate smaller edge location in order to reduce the network latency for time-critical messaging and computation, such as processing data from IoT devices. The global presence allows for operating responsive follow-the-sun services for a global audience, while it is less impactful on time-tolerant batch processing. A rich portfolio of turnkey services especially for data ingestion, processing and analytics is available at all hyperscalers. While often competitive in portfolio and pricing, failures in hyperscalers do occur frequently, often with devastating impact for large parts of the economy. Moreover, they are typically not well integrated into local research, innovation and supplier structures, leading to long-term issues with digital souvereignty.

Hence, a conscious and informed choice of the operational model with clouds, as a basis for DataOps, is crucial. More background information on three typical cloud scenarios is consequently given next.

4.5.1 Full application hosting

In this model, hypervisors, container engines and programming language-specific frameworks (e.g. web applications or function executors) are offered as managed services, ready to ingest custom program logic in the form of virtual machine images, container images or program code, respectively. This logic needs to adhere to certain conventions concerning port numbers, environment variables and lifecycle behaviour. Likewise, an application needs to be broken down into parts that fit into this environment, with appropriate glue in the form of triggers and messages in between. In a layered cloud environment, basic program execution is referred to as Infrastructure-as-a-Service (IaaS), whereas higher-level program management along with development and middleware services is referred to as Platform-as-a-Service (PaaS). The running application, serving multiple end users or tenants through a web interface or other interfaces, is then referred to as Software-as-a-Service (SaaS). However, all of these collective *aaS (or XaaS) terms are often used in a blurred manner in commercial practice and are merely a vague indicator of the service functionality and characteristics.

4.5.2 Partial hosting and on-demand offloading

Both in fully public cloud-hosted and in self-operated, private cloud-hosted scenarios, it might be useful to speed up program execution or achieve greater functionality by offloading crucial parts of the program and data to another cloud. Such offloading requires bundling the credentials of the secondary offloading cloud with the application code running either on the primary cloud or on cloud-attached devices such as mobile phones or edge machines. It

might also require workflow orchestration so that the offloading happens with the right context. Multi-cloud and cross-cloud frameworks such as Crossplane attempt to abstract away from the concrete cloud runtime locations. They facilitate programming in the large across execution technologies and providers, but are still emerging and neither required nor recommended for data scientists.

Similarly, data can be offloaded to storage services while performing the computation in a conventional form. This is especially useful for differential storage, in which data archiving happens across providers for maximised durability. The commonly used storage abstractions (FUSE, RClone) offer support for over 40 commercial cloud services and eliminate the network protocol differences between these services, presenting them all as unified storage resources.

4.5.3 Cloud backup

Almost the inverse to selected offloading of computation to the cloud is merely using the cloud as occasional backup for data. All primary data remains elsewhere, but sending encrypted data into a cloud storage service is cost-effective (ingress is usually for free), protective against data loss (although recovery egress will cost), and reasonably secure against data leaks. If infrastructure can be self-operated and hyperscaler service offerings are not needed, then this cloud usage pattern should be evaluated. Data can be synchronised to the cloud in intervals or based on events. Moreover, a trade-off between confidentiality and functionality can be achieved: all unencrypted data can still be processed by cloud services, for instance for analytics dashboards, and even some forms of encrypted data (using homomorphic, order-preserving and structure-preserving encryption) can still be put to constrained use without revealing too much. Lastly, anonymisation algorithms can be applied to maintain data characteristics while hiding the most concerning aspects, in particular Personally Identifiable Information (PII) from a privacy angle.

Repetition

1. Would a service offered on the IP address `127.0.0.1` be reachable from other computers?
2. A web service should be formally documented. Which language can be used for that purpose?
3. A latency-sensitive software function is to be deployed at massive scale to respond to vehicle movements. Would the deployment be more suitable in cloud computing or in high-performance computing environments?

Chapter 5

Applications and Tools

In this section, the foundational and conceptual knowledge conveyed in the first part of the book is put into practice in a local system context. This refers to a number of applications, primarily in the form of small composable tools and utilities that run locally on the interactive workstation of the data scientist or within the personal account on a remote server. These applications and tools support the exploration of the operating system and the juggling of data, models and code on that system as needed. Whenever appropriate, families of semantically identical or at least very similar tools are outlined first. To keep the text compact, at most one of the alternative implementations within each family is explained in detail. That does not imply that the alternatives are not valid choices depending on the scenario and surrounding conditions. Appropriate further reading pointers are given in selected cases to allow for making an informed decision about viable options.

5.1 Fundamentals

In order to work with local data processing and management tools efficiently, one has to understand a number of underlying concepts. Good tools are *optimised for one task* but also *configurable* and *composable* to be able to accomplish more complex tasks. Depending on the operating system, these concepts are more or less implemented on the operating system level or within third-party tools. Conventionally, Unix-like systems such as Linux and Mac OS X have had strong support with many pre-installed tools, whereas Windows has had limited support for many common tasks especially in networked environments. Nevertheless, all OSes evolve and, judging by their footprint of requiring multiple GB of storage space for installation nowadays, ship with an impressive number of tools out of the box. Generally, any functionality

already implemented in a tool and battle-tested in engineering scenarios saves time and effort to re-implement the same functionality, including in a higher-level programming language such as Python.

When interfacing with computers, the input modality or user interface is a primary concern. Tools are either *headless*, requiring no user interaction, or assume input and output in formats such as plain text (TUI, or Text User Interface), raster graphics (GUI, or Graphical User Interface), speech (VUI, or Voice User Interface) or, still rarely, gestures, eye movement or neuroelectronics. The history of computing has favoured text-mode tools, in alignment with textual programming languages, for automation and control tasks, which therefore form the focus of this chapter. Text refers to characters that are human-readable in the mastered languages and also human-writable, which comes with some challenges due to restricted keyboard layouts. The output of the text-mode interaction can be based on a raw line-based terminal where printing occurs sequentially but the cursor can be shifted as well as the terminal cleared. The interaction can also be done on the basis of a text-based navigation menu, with the background of text characters forming into rectangles that represent menus and input widgets. Foreground font attributes such as colours, underlining and bold face are also commonly supported by terminals to improve the text appearance. Limited support for mouse actions is provided by some text-mode applications, but typically the keyboard is the main input device. While the era of true text terminals (i.e. text-only combinations of screens and keyboards) has long been over in favour of high-resolution displays, the concept of terminals still exist today through terminal emulators, virtual terminals and virtual keyboards.

Such text tools, accessible through a terminal, work in a controlling operating system environment, typically a *shell*, that runs itself in a terminal as its main interaction point while putting itself into the background whenever a tool is in the focus. The shell completely supervises the tool lifecycle and facilitates composition.

A shell running as OS process permits navigating the filesystem and interactively entering the name of an executable on the search path. It then starts that executable as a subprocess, either in the foreground or in the background, with the working directory of the process set to the current directory of the shell. Tools might also be launched by other tools programmatically. While some tools strictly abide by the principle of being optimised for a single repetitive task, the behaviour of many tools can be adjusted to a certain degree. This can be accomplished statically before or during the invocation by parameterisation, through environment variables, or through configuration files. Moreover, some long-running tools allow for dynamic reconfiguration through OS signals, configuration file updates and other techniques. Command-line parameters, options and arguments are specified as a space-separated list after

the command name. They are then interpreted by the command depending on its implementation; for instance, a tool created in Python can refer to the system argument vector (`sys.argv`).

Once a tool is running, it can either remain quiet or provide text output to inform about progress and results. In addition to direct feedback from the execution in the form of standard output and error messages, many tools (especially those operating in headless mode) also produce log files that can be inspected after the execution to verify whether the invocation succeeded.

Most OSes support a superset of the shell and tools functionality defined in the 3rd volume of the IEEE Standard 1003.1-2017 "POSIX" (Portable Operating System Interface) that can be consulted for a more formal coverage.

5.2 Mastering Tools

5.2.1 Text-mode interaction

As text consists of individual characters, being able to read and type these characters in a fast way is a key skill in effective working with text tools. Reading and understanding character sets is a first step towards that. In Unicode, most characters have a single width, although some symbols from languages may have a double width, hence occupying twice the space as a single-width character. Text terminals are not always able to represent all characters depending on the chosen font (often replacing them by just an empty box □) and moreover may have problems with variable-width character sets. Legacy tools may also have problems with alphabetic characters beyond the Latin alphabet, although this issue has been reduced in recent years. Characters in Unicode are grouped into language-specific letters and symbols (A–Z, umlauts, accented letters and others), numbers, visible symbols including punctuation and mathematic operators (e.g. `<`, `;`) as well as invisible control characters such as backspace or enter.

With the keyboard, a limited set of characters can be entered directly by typing a key. More characters become available by combinations: ⇧(shift)+key, Alt+key, Alt+⇧+key, or multiple keys pressed in the right order. Multiple physical keyboard layouts exist to further complicate the input. For high productivity expressed by fast typing, it is important to understand these combinations, some of which are not shown on the physical keys themselves.

Care must also be applied when copying and pasting text from other sources such as web pages and PDF files. Often, they embed formatting and text modifications meant only for human consumption such as ligatures, ellipses and adjusted quotation marks (compare: „fi. . .“ and "fi..."). Search and replace patterns are then needed to make such text usable in the human-computer interaction. Pasting with Ctrl+Shift+V helps removing formatting.

Raw typing speed alone is often insufficient to be productive. Using macros or user-defined functions to avoid typing in the first place is often possible within shells, text editors and other tools. Moreover, interactive text input often allows for retrieval of the previously entered text for modification, which is another timesaver.

5.2.2 Types of tools

When choosing an appropriate tool, one should take a number of considerations into account. The first one is documentation. Good tools are documented with regard to what their purpose is, how they are configured and invoked (including examples), what error situations may occur and what standards they follow, if any. The second consideration is interface stability. Tools that have been around for a long time and are properly maintained by the OS distribution require less pre-configuration and are less likely to break in the future. This is important in the context of reducing technical debt when creating scripts for automating tasks.

In the next subsections, a number of representative tools from various categories are introduced. There are certainly valid alternatives to any of them; however, in order to get the job done, a data scientist is supposed to master at least one per category, and that one is introduced in sufficient detail. Several tools exist in multiple flavours, for instance, as stand-alone tools and as Python modules, enabling the re-use of functionality across working environments.

1. Operating system interaction: Shells, inspection and management of files, OS interaction, package and container management.

2. Data management: Synchronisation, version control.

3. Data processing: Text search, text processing and numeric processing, as well as visualisation.

5.3 Shells

Shells are command interpreters and process managers at the operating system level. In order to control and manage a system, but also to run and use complex infrastructure, it is essential to understand the basic role of local and remote shells. Interactive shells provide the command-line interface (CLI) as a specialisation of a textual user interface (TUI) to further tools and application programs. In contrast, shell command can also be invoked in batch mode. In this case, the commands are supposed to be written in a script file that is linearly interpreted, and interaction is only possible at the level of individual commands.

5.3.1 Overview on shells and terminals

Due to historic developments, there is no single shell. Rather, each OS has its own concept of a shell, and each shell has its own command set and language for automating the execution of tools. Windows has two main shells - Command Prompt (cmd) and PowerShell. Typing `cmd` in the OS start menu allows launching an instance of the former. These shells are both tightly coupled to the terminal window, whereas other operating systems maintain a separation. Mac OS X has the native Terminal.app that uses either Bash or Zsh as shells; although many users prefer iTerm that adds support for the Tcsh and Fish shells among other features. And Linux has a whole range of shells (e.g. Bash, Dash, Zsh, Fish, Tcsh, Ksh, Xonsh, Busybox-Ash, Yash), in addition to a plethora of graphical terminal emulators that make the shells accessible to their users. These emulators are often strongly related to the corresponding desktop environments, such as GNOME Terminal, XFCE Terminal, Konsole, ETerm and XTerm but can be used interchangeably. On a graphical Linux desktop, one either types the emulator's command name (`konsole`, `gnome-terminal`, `xfce4-terminal`, `xterm`, `Eterm`) or typically finds a menu entry for a terminal under System or Tools. Even on such systems, a text-mode shell terminal might be autostarted by default. Switching from the graphical environment to the text mode is then possible with a key combination such as Ctrl+Alt+F2. However, this should rarely be needed, especially with graphical terminal emulators that can be started in full-screen mode or switched to this mode via a menu entry or application-specific key combination, such as Alt+Enter in XTerm.

5.3.2 Local shell access with Bash

Bash[1] is one of the most powerful shells, acting as an interface between the user (or user applications), the virtual terminal and the operating system. It is one of the native shells in Linux and Mac OS X and is also available as an add-on package in Windows. In this section, Bash is explained in greater detail as one of the most capable and widely used interactive shells.

Bash commands can be given interactively at the *command prompt* or as interpreted files in the form of shell scripts. This is similar to Python, whose shell (i.e. interpreter) understands both interactive commands and scripts. However, the Bash language is different from Python in many ways. For example, built-in command names differ (`echo` instead of `print`), variable assignments must not use any spaces (`a=0`), and references to variables require a dollar sign (`echo $a`). Single-line comments, on the other hand, are given in similar form preceded with the hash or pound sign (`#`).

[1] Bash website: `https://www.gnu.org/software/bash/`

Attributed to the rich feature set of Bash is its startup behaviour, which is often too slow for non-interactive use, i.e. batch scripts. Therefore, alternative shells remain relevant, such as Dash for non-interactive shell scripts. The mix of shells might lead to confusion on the user side. A command executes well on the interactive terminal but does not run in a script. Often, this is caused by different shells executing the same command, with correspondingly different behaviour and outcome. As a rule of thumb, all shell scripts should be explicit about the interpreter in the form of a shebang line as explained below.

A brief introduction to the language and behaviour in addition to practical Bash handling is given in the work 'Bash Quick Start Guide' and reference usage documentation, such as the manual page for Bash (`man bash`), also hosted as an online copy.[2] This section only documents a few essential Bash commands and concepts to cover the occasional use, without claiming to be a full shell programming guide.

The canonical form of interactive usage is `bash`, leading to a new shell process being instantiated. Alternatively, the shell is invoked as a non-interactive wrapper around a command, i.e. `bash -c 'command to be executed'`, where commands may range from simple executables to compound commands with input-output redirection, boolean logic and other shell execution facilities. A special form is the login shell, which is invoked automatically upon text-mode login to the system, either at a local text-mode login prompt or over the network using a terminal emulator. Login shells are interactive but ready different configuration files upon invocation. The startup of the shell can thus be customised through invocation-specific configuration files that are themselves shell scripts. For login shells, this is primarily the system-wide file `/etc/profile`, complemented by the per-user file `~/.profile` and further Bash-specific files in the home directory (`.bash_profile`, `.bash_login`). For non-login interactive shells, notable configuration takes place in `/etc/bash.bashrc` and the `.bashrc` file in the user's home directory. This file can source other files, and this is supported by convention with `.bash_aliases`. In contrast, non-interactive shell processes do not read any default configuration.

Bash scripts marked as executable in their file metadata (explained below) can and should contain a first line pointing to the executable of the shell to execute these scripts with exactly that shell. This is called a shebang line. For instance, a script called `test.sh` might be marked with `#!/bin/bash` so that running `./test.sh` works and shows up in the process list despite the shell script not being an executable in the OS sense of the word. This first line is treated as a comment by the shell itself but instructs the OS loader to invoke the right shell.

Interactive Bash processes require a controlling terminal, either a native text-mode terminal of the operating system or a graphical terminal emulator.

[2] Bash manual page online: `https://man7.org/linux/man-pages/man1/bash.1.html`

These interactive shells keep track of the current working directory and permit navigation with built-in commands such as `cd` (change directory). The command prompt is configurable but in most cases shows the working directory for reference. The special character ∼ (tilde) refers to the current user directory, e.g. `/home/user`, and ∼`otheruser` to the home directory of other users.

Commands to repeat in alphabetic order: bash, cd (built-in)

5.3.3 Bash variables

Variable names in Bash are case sensitive. They start with a letter (restricted to ASCII) or underscore and can contain further letters, underscores and digits. Variables are dynamically typed and created by assignment to a variable name by using the equal sign without surrounding spaces and without spaces in unquoted values. Exemplary assignments are `var=123`, `var=abc` and `var="abc def"`. Both single and double quotes can be used, with the difference that other variables referenced within double quotes are substituted by their values, whereas single quotes force the verbatim assignment. The only exception is the assignment of special characters in the form of `var=$'\n'`.

Assigned variables can be used with the dollar sign prepended, such as `$var`. The most common case shows their value interactively with the shell built-in command `echo`, as in `echo $var`. While the echo command tolerates an empty argument in case the variable is not set at all, other commands do not necessarily tolerate the same. To enforce passing an empty argument instead of no argument in this case, the recommended notion is enclosing the variable in quotation marks, resulting in the instruction `echo "$var"`. Another difficulty is that, in a string context, Bash would sometimes not know the boundaries of a variable name, as in `echo "$numberhouses"`. The boundaries can then be supplied explicitly, as in `echo "${number}houses"`.

Bash scripts and functions can be parameterised; while `$0` refers to the script or function name itself, `$1` and following arguments contain the parameters. They can be tested for being empty or not with an if-clause involving for instance `test -z "$1"`. In that case they should always be enclosed in double quotes, because the variable may not exist, leading to a syntactically incorrect test statement. In Bash, the test instruction can be replaced by square brackets in conditional clauses. For instance, `if [ ! -z "$1" ]; then ...; fi` only executes a block of code if the first parameter has been supplied.

Bash processes also keep track of the environment through *environment variables*. Any process spawned from the shell inherits these exported variables so that custom program behaviour can be triggered by setting up appropriate variables. This differs from regular local variables that are only valid within one shell process. A simple assignment of an environment variable may read

like `export a=0`, although exported variables by convention use uppercase names. All variables can be shown with the `env` command, or alternatively shown and filtered with `printenv`. Among the ones often referred to are `$USER`, `$HOME` and `$LANG`, representing the current user's identity, home directory and language setting, respectively, as well as `$PWD` and `$PATH` further explained below. Rarely, there are recommendations to set `LD_PRELOAD` to inject override functionality into compiled applications, such as `eatmydata` to avoid file system syncs, `fakeroot` to not let privileged operations fail or faking the system time; however, this dynamic reprogramming must be used with care. To temporarily set an environment variable for one particular process without affecting the shell or other processes, the variable assignment can precede the command, as in forcing a particular language: `LANG=en_US <command>`.

While environment variables are a portable concept and also work in shells other than Bash, there are additional Bash-internal variables predefined by the interpreter instance and partially affecting the programming behaviour of the shell itself. Examples include `OSTYPE` and `HOSTTYPE` giving information about the operating system, `EUID` containing the effective user id of the current process and `IFS`, the internal field separator used when reading from sequences of data. These variables are not shown in `env`. To nevertheless include these variables but also other internal definitions such as functions, and to see all entries in the entire namespace of the running shell, the `set` command can be used. Shell-internal variables are explained later in the section on shell programming.

Special variables exist in Bash to retrieve the current shell process identifier (`$$`) and its parent (`$PPID`) and to generate a random integer number in the range 0–32768 (`$RANDOM`).

> Commands to repeat in alphabetic order: echo (built-in & executable), env (built-in), export (built-in), printenv (built-in), set (built-in), test (built-in)
>
> Environment variables to repeat: $$, $HOME, $IFS, $LD_PRELOAD, $LANG, $PATH, $PPID, $PWD, $RANDOM, $USER

5.3.4 Bash commands

The imperative vocabulary of shells consists of both built-in and executable commands, where built-in commands are internal to the shell (part of the default vocabulary) or provided by the user within the shell programming environment, and executable commands refer to programs found in the search path. A number of executables are always installed by default on any system, whereas, sometimes, useful tools exist but must be regularly post-installed,

referred to as non-default or external commands. This is especially true for services (daemons) and other middleware.

The search path for executables, expressed as environment variable `$PATH`, is interpreted as a list of absolute and relative directories separated by colon (`:`). New entries can be prepended or appended to it as necessary. The order of evaluation is from left to right. As soon as an executable matching a command name not available as internal command is found, it is taken as a match. Hence, this paths list has similar semantics to the variable `sys.path` for searching modules to be included in Python. Notably, the default path does not contain the current working directory by default (`.`), and therefore executables not on the search path need to be addressed by relative or absolute path, such as: `./myprog` or `/opt/myprog`. Alternatively, the paths list can be extended to include the current directory (`export PATH=.:$PATH`). In that case, running just `myprog` works. Attention should be paid to this because it is at risk of changing behaviour when accidentally a globally installed program is shadowed by an executable produced in this directory. With the `which` command, the first matching path for an executable is informed.

The most trivial built-in is the colon (`:`), a no-op command that does not do anything but having command semantics, similar to `pass` in Python. POSIX defines around 20 built-ins while modern shells support a few more. Programmable shells also permit the creation of custom internal commands. Bash in particular allows two kinds of custom commands: simple aliases (with the built-in `alias` command) and arbitrarily complex functions. The resolution order matters so that sometimes external commands are shadowed by built-ins with slightly different behaviour (`echo`, but also `time` and `kill` introduced later). For instance, when talking about `echo`, this might refer to the shell built-in explained before or to the namesake executable on the default path, which is not much different (`/usr/bin/echo`), or potentially a self-built executable in the current working directory in case that is part of the search path. The `type` command, similar to the function with the same name in Python, tells about what type a command name is of (built-in command, user-defined function, alias, external command/executable or reserved name). A brief documentation for built-in commands may be shown with the `help` command.

Commands are entered on a per-line basis at the command prompt. Command names may be autocompleted with the Tab (tabulator) key (↹). If the completion would be ambiguous because multiple commands with the same prefix exist, the Tab key needs to be pressed twice to show all of the candidates. For instance, typing `e<tab><tab>` shows `echo` among other command options. After completing a command with the Enter key, the command is executed and the result is shown before returning to the prompt. Previous commands may be retrieved in order by pressing the Arrow-up key. Search-

ing through previous commands is possible by pressing Ctrl+r followed by a search term. When composing the input at the prompt, further quick navigation is possible with, for instance, Ctrl+a and Ctrl+e to jump to the beginning and the end of the input line, respectively. With those few keyboard strokes, entering commands becomes efficient and the shell becomes an indispensable tool for automating tasks.

Complete commands in Bash consist of the case-sensitive command name, either built-in or an executable, and a number of options, parameters and arguments, all separated by spaces and otherwise defined in an application-dependent way. Values with spaces can be quoted with ' (single quotes) or " (double quotes). The quotes may be combined, but if single quotes are used in the outer scope, then variables are not expanded within it. Arguments with placeholders (`?` for single characters, `*` for multiple characters) are expanded based on the content of matching files and directories in the underlying filesystem, through so-called globbing, but only if there is at least one such file or directory. Otherwise, the placeholders remain in place verbatim, which is often not the intended behaviour. There is also a limit to the size of the argument list. Hence, copying (`cp *.jpeg myfolder`) might fail if the number of files is too high; in that case, an iteration over all files with a `for` loop or with the `find` command must be used. Overall, shell globbing is a powerful feature but must be used with care due to the mentioned and often not obvious limitations.

Multiple independent commands may be chained for serial execution by `;` (semicolon), although that is not recommended as it impedes readability compared to placement across multiple lines. With the `parallel` wrapper command, multiple instances of the same command or other independent commands can also be run at the same time. Lastly, the pipe operator `|` is able to chain dependent commands such that there is a data stream flow from the first to the last while they execute in parallel.

An example invocation combining both piping and parallelisation is to shorten the execution time by converting multiple photos from the digital camera to a lower resolution. It could be achieved as follows: `ls P*.JPG | parallel convert -scale 1200 {} S{}`. As a result of involving the external conversion tools, each image (e.g. `P1020.JPG`) ends up downscaled and renamed (`SP1020.JPG`). Drawing from the way `parallel` works, one can concluded that external shell commands can be divided into two groups: regular commands, and wrapper commands that execute commands given as parameters. Among the more useful wrapper commands beyond the ones already mentioned are `stdbuf` to change especially the output buffering behaviour and `timeout` to run a command under a time barrier.

All commands are executed in the foreground, blocking the shell from proceeding, but they can be put into the background either at invocation time by appending `&` (ampersand) or after interactive invocation by pressing Ctrl+z

(suspend) followed by the built-in command `bg` (background). Likewise, processes can be brought to the foreground again with `fg`. Applications can be cancelled and terminated with Ctrl+c. In practice, this job control is only useful for batch processing, whereas some programs are meant to run interactively in the foreground as they take over the entire terminal and do not leave any output trace after termination. Batch output can also be redirected to a file, assuming the specified file location is writable for the current user. Writing and overwriting the file is achieved using the `>` operator for regular output and `2>` for error output, mirroring the ability to redirect input with `<`. Appending to the file without destroying previous content is also possible with the `>>` operator. If output should still be visible while at the same time be captured to a file, piping can be used as follows: `command | tee <file>` for overwriting, and using `tee -a <file>` for appending.

Subcommands can be executed and their output captured with the backtick character (‘), as in: `command ‘subcommand‘`. The subcommand is then executed first, and its output is placed in lieu of the backticked placeholder, effectively executing the first command with the subcommand output as argument.

Bash contains built-in facilities for integer arithmetics, for instance, `echo $(($a+1))` and basic support for advanced data types such as dictionaries. For more precise calculations and more versatile data structures, external tools must be used. Finally, Bash processes can be quit with the `exit` command, like Python but without the parentheses, or alternatively the keyboard combination Ctrl+d.

The following listing shows exemplary shell commands with an explanation.

```
echo "a b c" # output the string "a b c" to the terminal's
    standard output channel
sleep 5       # do nothing for five seconds
sleep 5 &     # do nothing in the background, while
    liberating the command prompt

ls -l /etc > ~/conffiles.txt # create file with list of
    system configuration files
```

Like the built-in help command in Python, it is often possible to obtain at least basic documentation on commands via manual pages. The representative command `man ls` documents all possible parameters for listing files and directories. For built-in Bash commands, the `help` command is used instead. Finally, when it becomes hard to keep the overview, the `clear` command can be invoked to clear the screen. In case the terminal is severely messed up and no longer produces linebreaks, the `reset` command can help to bring its state back to order.

Commands to repeat in alphabetic order: bg (built-in), clear, cp, fg (built-in), help (built-in), man, parallel, reset, sleep, tee, type (built-in), which

Environment variables to repeat: $PATH

5.3.5 Remote shell access with OpenSSH

Oftentimes, the data scientist's local workstation environment is considered unfit for a certain task due to general resource shortage (disk, memory, processors), lack of specialised resources (especially GPUs), outdated or unsuitable OS, or the need to collaborate among multiple people. In these situations, a dedicated physical computer might be set up or a virtual machine might be instantiated at a virtual machine provider. This machine then serves as a remote machine to which a connection can be established, linking it with the workstation by allowing cross-machine file access and tool execution.

An interactive and secure remote shell connection to any server can be established using the SSH (Secure Shell) protocol, most commonly in the form of *OpenSSH*. This stateful application creates a remote work session in which it takes control of keyboard input on the local (client) machine and relays it to the remote (server) machine, more specifically to the configured login shell of the chosen user, while relaying back the responses. The OpenSSH client has over time become a standard tool for remote operations on all major operating systems – Linux, Mac OS X and Windows. Apart from text-based sessions, it also permits the forwarding of graphical applications on pairs of systems supporting the X11 protocol, such as Linux.

The canonical form of usage for a text session is `ssh <user>@<server>`, using the default port number (22) and default negotiation of encryption parameters, the so-called ciphers. This command either asks for a password, or in case an SSH key is used for authentication, it may or may not ask for a passphrase, depending on how the key was created.

Graphical counterparts to the SSH CLI are available, for example, `putty` as utility on Windows formerly recommended over many years. However, they might not be suitable for automation in data science-related shell scripts. In case the standard OpenSSH CLI client is not available on a Windows installation, it can also be installed in Powershell easily with administrator privileges: `Add-WindowsCapability -Online -Name OpenSSH.Client~~~~0.0.1.0`.

Hence, no matter what shells are installed locally on machines, remote shell access permits standardising shell scripts and other shell-based automation across teams in a portable manner.

Authentication is performed using passwords or better using a public/private key pair that must be first generated on the client. The private key remains on the machine the user is working one, while the public key can be

deployed on the remote machine, including in several collaboration services beyond SSH itself. The usual command to generate a key on the local workstation is `ssh-keygen -f <filename.key>`. The command can be simplified to `ssh-keygen` upon first use, resulting in the key stored in the default location. On Unix-like systems, the private key path is `~/.ssh/id_rsa` whereas on Windows it is `C:\Users\<user>\.ssh\id_rsa`. The corresponding public key has the same path, but with `.pub` appended to the file name. A key can be protected with a passphrase, although that can also be omitted, especially when automated batch authentications are planned. Losing a password-less private key would, however, constitute a grave security risk, and the key needs to be handled with extra care to prevent that from ever happening.

With OpenSSH, the public key must then be placed into the file `~/.ssh/authorized_keys` on the remote account with locked down permissions by the account administrator or, if password-based logins are still enabled, by the user. The permissions require both the `.ssh` directory and the file to be only accessible by the user itself including automatically the superuser. Moreover, unless the default path is used, the key needs to be specified as identity upon each connection (`-i`) or the configuration file (`.ssh/config`) needs to have an `IdentityFile` setting for the affected `Host` entry.

Files are transferred via the SSH protocol with the `scp` command, meaning secure copy, in the form: `scp <filename> <user>@<server>:<path>`. The path can be omitted in case the user's home directory is the target. SSH also supports a secure variant of the File Transfer Protocol (FTP) with the `sftp` command. After logging into a system with `sftp <user>@<server>`, typical FTP commands can be used to copy files back and forth and to compare the presence of local and remote files.

Usually, the first file to transfer from the workstation to the remote machine is the SSH public key to prepare it for use. This procedure is explained in the following listing. All commands around copying files are explained in greater detail in a later section.

```
# First, copy the public key to the remote machine, into
    the user's home directory
scp id_rsa.pub user@machine:
# Then, log into the machine, and perform remaining steps
    there
ssh user@machine
# Create the .ssh directory with the right permissions; the
     mode 700 is explained further down
mkdir -m 700 .ssh
# Register the public key file
chmod 600 id_rsa
mv id_rsa.pub .ssh/authorized_keys
```

The commonly used flag `-r` is used for recursively copying entire directories. The secure copy commands are explained in greater detail in the section on file management. A remote shell can also be quit by terminating the shell with the `exit` command or by Ctrl+d.

More information on OpenSSH usage is available through the auxiliary article 'SSH: a Modern Lock for Your Server?' as well as through online documentation.[3]

> Commands to repeat in alphabetic order: exit (built-in), scp, ssh, ssh-keygen

5.3.6 Advanced shell management with Screen and TMux

Sometimes, the interactive use of applications, either shells or other applications spawned from them, stretches across long sessions beyond the login period. Due to occasional disconnects when using remote sessions caused by the SSH configuration or external network events, the login period can sometimes be quite short, and any tool running for more than a few seconds is at risk of either continuing its execution in the background or even of termination when no special protection is applied. Screen is a tool which extends remote shells with the possibility to reconnect to them later without losing the session. It achieves that by running as invisible layer in between the application and the shell. The shell runs Screen which in turn spawns a long-running background process, and that Screen process runs the tool command. It should be noted that Screen thus protects against disconnects, but not against restarts of the server itself. Evidently, in such cases the tool must be further protected with automated restart and internal checkpointing capabilities.

A basic invocation is `screen <command>`. Screen runs as long as the command itself, but a preliminary detachment is possible with the keycode sequence Ctrl+a;d (i.e. first pressing the control key (Ctrl) and the letter A without shift (a) simultaneously, then releasing the keys, then pressing d). The detachment drops the user back into the shell Screen was started from. Closing this shell has then no effect on the screened command execution. To re-attach, all running Screen sessions can be shown with `screen -list` and a particular session can be chosen and restored with `screen -r <session>`. Default session names include the terminal number and the local host name. Screen sessions can be invoked in the background in the first place, even with human-recognisable names, by adding the parameters `-d -m -S <name>`. Many additional invocation options and key codes for managing sessions exist. The authoritative documentation for Screen is available online[4],

[3] OpenSSH manual: `https://www.openssh.com/manual.html`

[4] Screen documentation: `https://www.gnu.org/software/screen/manual/screen.html`

and the manual page is also sufficiently useful for further customising the use of the tool.

On larger displays, it may also be desired to split up the terminal estate into two or more shells. That way, the output of two commands can be compared, or one command can be inspected while a shell is provided at the same time to further control that command, among other scenarios.

While Screen has some support for splitting sessions, a small tool not necessarily bound to background sessions might be useful in this case. TMux, the terminal multiplexer, can do just that. Its canonical invocation is just `tmux`, launching a subshell that behaves like the parent shell but accepts special key codes to manage screen splitting. Commands to a running tmux session are given by key sequences starting with Ctrl+b and followed by another key or combination of keys. Most importantly, Ctrl+b;% creates a new horizontally split pane (" for vertical split), and Ctrl+b;Ctrl+o rotates through the panes. Each pane runs a shell by default and can simply be closed by the command `exit` or Ctrl+d.

Commands to repeat in alphabetic order: screen, tmux

Repetition

1. How can a user log into the server X with account name Y?
2. How can the same user transfer a file Z into his or her home directory on the server?
3. The user would like to work on a text over many hours. Which command sets up a suitable long-running shell session?
4. Which wrapper commands to launch other commands exist?

5.4 Useful shell tools

Data scientists who are familiar with the shell as a working environment soon understand why it is called a shell: It is just a shell for a lot of tools that permit interacting with the system and with data. Like a precious pearl, the interest soon shifts to those tools. In this section, a lot of standard tools are introduced. The ambition should not be to remember all of them, at least not at once. Rather, the section makes an attempt to group similar tools together and explain their relationships, so that looking them up later becomes a systematic and efficient process. For that matter, seven groups of tools are distinguished.

The first two sections explain basic overview and exploration tools for the underlying hardware resources and the operating system, respectively. This is

followed by three groups of tools related to time and event handling, to data organisation through files and directories, and to data creation and modification. Two more sections then dive deeper into issues around networking and system administration, both of which form the basis of distributed DataOps in practice.

5.4.1 Hardware resources exploration

In a dynamic world where users roam across different local and remote systems, getting an overview of the current capabilities on the operating system and underlying hardware resource levels is often the first step. The commands `free -h`, `df -Th` and `lscpu` (or more verbose, `cat /proc/cpuinfo`) are used to convey information about the available main memory, free disk space per file system and CPU resources, respectively. They are usually called first on a new system to verify the ability to work. In greater detail, the `free` command differentiates between main memory and, if configured, disk-based swap space as resources to store volatile process data, and indicates the amount of available, currently used, and free memory. Memory usage occurs as the sum of all usage per process, with some being more memory-hungry than others. The free value often tends to be quite low, almost going towards the zero line, due to the OS keeping used memory pages (blocks of memory) around as cache and buffer memory and using some pages as shared area between processes. Therefore, the tool also informs about the net free value, which is often higher and the value of interest for checking if sufficient memory would be available when freeing up all disposable pages. While the default unit of the output numbers is KiB and the output could be processed by automated scripts, the parameter `-h` is meant for human consumption and adapts the unit to what makes sense for a number, such as MiB or GiB. The option `-h` is not universal and often implies the invocation of help, but a number of system-related tools use it to express human-friendly output.

The `df` (disk free) tool is such an example. It is the equivalent of `free` for persistent data stored across multiple storage media and storage-related block devices, potentially across an even higher number of partitions. Its often-used option are the mentioned `-h` for human-readable output and `-T` for displaying the partition types, effectively the file systems in use in each partition. This tool shows not only available and used disk space for each partition but also where that partition is mounted in the file system hierarchy. A system consists of at least a persistent partition at the root directory `/`, using file systems such as *ext4* or *xfs*. Typical systems furthermore include a safety-critical and therefore separated persistent boot partition `/boot` as well as OS-specific internal volatile partitions outlined below.

The command `lscpu` (list CPU specifications) informs about the processor architecture and performance, the number of processor cores, cache sizes,

hardware virtualisation support and similar CPU information. In contrast to the other two tools, its output is human-readable by default. For automation use, `/proc/cpuinfo` is consulted or `lscpu -b` is invoked to show all sizes in bytes, or even `lscpu -j` to output JSON-formatted data.

To get further information about system resources, a family of `ls`-derived list commands similar to `lscpu` exist. For instance, information about peripherals can be obtained with `lspci` for PCI-connected internal peripherals such as graphics adapters, disk and network controllers, audio chips and non-volatile memory. Likewise, especially for workstations, `lsusb` informs about the USB hub, devices connected to it, often including the notebook webcam by default, and plugged in storage media. Not all block devices may be mounted. The `lsblk` command shows the hierarchies of volumes and partitions and correlates them to active mountpoints.

Basic information about available network adapters can be retrieved by reading the file `cat /proc/net/dev`. A more comfortable management of network interfaces is explained later. With the covered commands, the capabilities of all important computing resources (CPU, RAM, disk, network, peripherals) should be known on any system, and the exploration can proceed further into the OS-specific processes and structures.

Commands to repeat in alphabetic order: df, free, lsblk, lscpu, lspci, lsusb

5.4.2 Operating system exploration

At the top of the hardware resources, `uname -a` (Unix name with all information) gives basic information about the running operating system kernel. A more challenging question could be which OS flavour or distribution is being used in case additional software needs to be installed. This is explained in the system administration section below. The name of a system is not sufficient to see what it does. Revisiting the `df` output, several internal partitions mounted in-memory without relation to physical block storage become evident. Some of them are using *tmpfs* as in-memory file system, such as `/run` for temporary data storage of services and `/dev/shm` for named shared memory areas. While the `df` output is human-friendly, it may omit information about some more internally mounted file systems, and hence the `mount` command gives a complete but less readable information about all file systems in use, all with highly exotic and system-dependent virtual file systems. This includes `/sys` for information on the hardware, `/proc` for running processes and `/sys/fs/cgroup` for process isolation among many others. Additional file systems may be mounted into the hierarchy from block devices with `mount -t <type> <device> <mountpoint>`, alternatively from images as loop mounts with `mount -o loop`, or from a directory with `mount -bind`.

More commands similar to `lscpu` exist for revealing OS structures. For instance, `lsipc` gives a statistical account of inter-process communication, with `lsipc -m` drilling down into shared memory areas. Information about open files and network connections is given with `lsof`. The more active a system is with running processes and services, the longer the output of these commands gets. They are useful especially for debugging, to find out which process has opened which file among other questions.

The main memory and disk-free commands in the previous section include information about the current resource utilisation, whereas the CPU-related statistics only show static capacity information. Going deeper into knowing the running processes and their computational needs requires process monitoring. There are tools to monitor all resources combined over time, such as `vmstat 1` showing memory and processor utilisation on a per-second basis, but their output is not easy to interpret and moreover does not reveal the troublesome processes that might cause a CPU to be overloaded.

Therefore, process-level monitoring is important to find out about the CPU utilisation and about the activities managed by an operating system in general. The command `ps` and the related `pstree` show the already running processes along with basic statistical information. Typical invocations are `ps wxf` to list all of the user's processes, and `ps wwauxf` to list all processes of all users with all arguments. Evidently, this comes with drawbacks such as verbose output and lack of highlighting the CPU-intensive processes. A live view can be performed with `top` or its fancier cousin `htop`. Both tools take over the terminal and need to be terminated with the key Q. In `top`, CPU-intensive processes are listed first, yet with the M key this can be changed to memory-intensive processes. In `htop`, process management works in a menu-based way. One or multiple processes can be marked with the space key (␣). To terminate selected processes, the F9 key (i.e. Fn+F9 on many notebooks) is pressed, offering the selection of a signal to be sent to these processes, by default the TERM signal that can be confirmed by pressing the Enter key.

In case a stray process is detected and must be terminated before re-invoking a command, the `kill` command outside of `top`/`htop` can be used for that purpose, taking the PID (process id) as parameter. This command sends a signal to the OS asking it to terminate the process, which works if the process is owned by the user sending the request or of course when the super-ruser sends this instruction. The default signal is *TERM*, asking for graceful termination while giving the affected process the ability to perform some last work or even block the request. A forced invocation would happen with the *KILL* signal, as that can not be blocked, via `kill -KILL <pid>`. Further useful signals from a user perspective are *STOP* to pause a process and *CONT* to continue its execution, while many other signals are sent by the OS itself to be handled by the process. Applications can implement handlers for interceptable user-defined signals (*USR1*, *USR2*), for instance, as Python methods.

If multiple processes are running with the same name, their PIDs can be obtained with the command `pidof <program>`. For example, `pidof bash` shows the identifiers of all active Bash processes, including both executed scripts and interactive sessions. With options such as `-s` (single shot) or `-q` (quiet, only report existence of at least one process via exit status), this tool is useful in shell scripts to detect for instance stray instances of a program, especially in conjunction with PID files, and to force termination of all instances in upgrade scenarios. Combining the functionality of `pidof` and `kill`, the `killall` command can send signals including soft and hard termination to a group of processes. Detailed statistics for a single process may be shown with the `prtstat <pid>` command.

The command `hostname` conveys information about the host's own internal name, which is typically reflected in the output of `uname -a` and in the command prompt appearance as well. As outlined in the description of networking concepts, the file `/etc/hosts` can map IP addresses to names for other hosts in the format `<ip> <hostname>`. On Windows, the corresponding file also exists and is located in `C:\Windows\System32\drivers\etc\hosts`, and on Mac OS X in `/private/etc/hosts`. Typically, entries in the hosts file override DNS queries. Often, though, the local IP address `127.0.0.1` resolves to `localhost` whereas the name given by `hostname` is unknown due to a missing entry in the mapping file. This might cause problems later working with services that report unresolved hostnames. The solution is to add both forward and reverse mappings to that file and update them in case the host gets renamed. To work further with the mappings, the `host <ip>|<name>` command can resolve names to IP addresses and vice versa but usually does not consult the hosts file unless this is configured in the file `/etc/resolv.conf`; the command `getent hosts` is more appropriate in that case. The `ping <ip>|<hostname>` command may be used to check for reachability of a host. It always consults the mapping file, although its communication via ICMP might be blocked by that host. No tool is perfect, and the necessary trade-offs show especially in attempts of debugging network issues..

A related command to convey OS identitity information is `whoami`, showing the current user that is likewise reflected in the prompt, along with `id` showing more information about that user, in particular group memberships. This encompasses the primary group by name and numeric identifier (*GID*), but also secondary groups. Moreover, the `uptime` command shows how long the system has been running and under which load. One may even maintain the history of all uptimes with `uprecords`. The related command `who` shows all users currently logged into the system, complemented by `w` giving more details and combining it with the uptime functionality, `last` showing the full history of previous login sessions, and `lastlog` informing about the most recent login of each user on the system. This typically includes special-purpose system users created to isolate more complex applications who obviously never log in.

With the additionally covered commands, the exploration on the OS level is complete. All essential configuration settings and actively running processes are known, and further tools can be used to perform productive work related to data, time and space (locations within the file system).

> Commands to repeat in alphabetic order: host, hostname, id, kill, killall, last, lastlog, lsipc, lsof, mount, pidof, ping, prtstat, ps, pstree, top/htop, uname, uptime, vmstat, w, who, whoami

5.4.3 Time- and event-related commands

The command `date` outputs the local date and time down to second precision according to the local conventions, consisting of both language and timezone. The language and associated culture-specific date format can be adjusted without affecting the time, as in: `LANG=en_US date`. The value of the environment variable `LANG` in this context is either just a language code (e.g. `en` according to the international norm ISO 639-1) or a language code combined with a country code (e.g. `US` according to ISO 3166-1 Alpha 2). The timezone can be set either to the canonical universal coordinated time (UTC) with `date --utc`/`-u` or with a timezone environment variable such as `TZ=Europe/Lisbon date`. The file `/etc/timezone` contains the system-wide timezone setting. Human-readable times can be converted into numerical timestamps with `date +%s`, and converted back from those timestamps with `date -d @1700000000` (short for for `--date`). These 32-bit integer timestamps refer to the number of seconds elapsed since the epoch on January 1, 1970. In other contexts, such as in Python's `time.time()` functions, the timestamps also appear with sub-second precision, indicating a microseconds fraction although the actual precision depends on the CPU quartz and is likely in the milliseconds range. In the shell, the closest equivalent is `date %s.%N`, indicating nanoseconds with the same caveat. Other standard notations of dates can be produced in the sorting-friendly ISO 8601 YYYY-MM-DD format with the shortcut `date -I` and in the human-friendly DD.MM.YYYY format with the string formatting `date +%d.%m.%Y`. Several more placeholders exist in the `+%` syntax, making custom formatting trivial.

To measure the relative execution time of commands as delta between absolute start and finish times, two timing facilities are available. The first is the Bash wrapper built-in `time` showing real elapsed time of a command with sub-second precision such as `time sleep 2`. The second is a more portable and sophisticated wrapper command with the same name that is shadowed by the built-in, and thus must be referenced explicitly, such as: `/bin/time sleep 2`. When supplied the option `-p`, its less readable output matches that of the built-in, providing portability also in that sense.

The performance of program execution as time-based metric is inherently resource-dependent. It is not trivially possible to slow down a processor or a disk in case certain situations should be simulated in a controlled environment. Per process, the priority can be influenced with the niceness level through the wrapper command `nice -N` with N between -20 (highest priority) to 19 (lowest priority). However, this only influences the relative priority and depends on the chosen scheduler, by defaut the Completely Fair Scheduler (CFS) in Linux. The `cpulimit` command (that needs to be installed separately) is helpful in this regard. It can act both as a wrapper command for new processes and to reconfigure running processes identified by a PID. With `cpulimit -l 5 <program>`, it would execute a program with 5% allocated CPU share.

On the networking level, throttling data transfer speeds is possible by introducing Sluice into the pipeline, which is another program to be installed separately beyond the typical base installation. For example, `cat <file.txt> | sluice -r 100 | ...` ensures that all data arrives at the end of the pipeline with a constant rate of 100 bytes per second. The throttling mechanism works with dynamically sized buffers, hence high speeds may lead to not all input data eventually arriving.

To run a command in regular intervals and compare the output in an interactive session, it can be watched with another wrapper: `watch -n 1 ls -l`. Complex commands should be quoted, as in: `watch -d -n 1 "pstree | grep x"`.

Non-interactive scheduling of commands based on absolute days and times can be achieved with Cron, or based on relative times pragmatically by a prepended sleep. Interacting with Cron primarily works by editing its plan (per user or superuser) with the `crontab -e` command, using the default text editor as described in the section on editing files. This requires discipline to produce a file in the right format, consisting of lines that are either comments (starting with `#`) or scheduled commands consisting of a time pattern and the full command line or environment variable settings. Viewing (listing) an existing plan is possible with the corresponding `crontab -l` command. A rarely used and rather destructive option is `crontab -r` to remove the plan for a user entirely.

The use of Cron requires understanding that as it runs as a central daemon, its time zone and path settings might be different from the user environment. To mitigate these issues, the standard Cron implementation supports setting and overriding environment variables (`PATH=...`) in addition to scheduled commands (e.g. `* * * * * <per-minute-command>`). The five time specification fields refer to the minute, the hour, the day of month, the month (counting from 1) and the day of week (starting with 1 for Monday but also accepting 0 for Sunday). An asterisk signals an unconditional invocation. Cron invocations are logged with a *CRON* tag into the system log file `/var/log/syslog`, which is useful to consult if a command did not run although it should have.

The behaviour of Cron is that missed schedules, often due to system downtime, are not repeated, and neither are commands that temporarily failed, for instance, due to a network transmission problem. Another restriction is that fine-grained scheduling on a per-second basis is not possible, unless with the workaround of using multiple invocation with sleep prepended if the interval is a divider of the 60 seconds of a minute. There are more advanced but not yet as widely installed alternative implementations such as `fcron`, `xcron` or `anacron` that overcome at least some of those limitations, although sometimes by introducing additional restrictions. If SystemD is available, then timer units can also start service units on specified times including with second resolution. SystemD is described later on in the system administration section.

When, instead of being triggered by time, events should rather run in response to certain general events, it is suitable to have a monitoring command that blocks until the event happens and then lets the next command run. Specifically for file system events, running `fsnotifywait -e modify /tmp/myfile` would block until a content modification happens with the specified file. This functionality can be used to trigger data processing after data arrival in an event-driven way.

In summary, the shell offers a plethora of tools related to retrieve absolute and relative (delta) time information and scheduling. Some of those might not be enough for production scenarios, and therefore especially for complex workflow scheduling there are alternatives that are discussed in later chapters of this book.

> Commands to repeat in alphabetic order: crontab, cpulimit (external), date, fsnotifywait, nice, sleep, sluice (external), time (built-in & executable), watch
>
> Environment variables to repeat: $LANG, $PATH, $TZ

5.4.4 Managing data in files and directories

The previous groups of tools were either command-centric or produced minimal amounts of transient data. Retaining the data and organising structures to maintain and use the data become important complementary activities that require a thorough understanding of the file system layout in operating systems including data and metadata. The subsequent paragraphs thus provide a guide to working with nested directories and files.

Directories are organised in trees of arbitrary depth, with files being leaf nodes. Each directory contains two pseudo-entries: a reference to itself (`.`) and a reference to its parent directory (`..`). This is even true for the topmost directory (`/`) for which both references are identical. The path to a leaf node flattens this tree structure. The leaf node name itself is referred to as

basename, whereas the remainder of the path leading to it is called dirname. Hence, the namesake commands `dirname a/b/c` and `basename a/b/c` would display `a/b` and `c`, respectively.

On most file systems, metadata on both files and directories contains of three sets of dates (creation, last modification and last access). This means that reading files causes write modifications on the file system, which might affect the underlying device's wear level. Apart from that association, reading and writing are thoroughly separated through permissions, which is the next information kept in the metadata. Finally, the ownership, the size and the name of a file or a directory as well as extended attributes complete the metadata.

The size of a directory represents the number of files and subdirectories contained in it. The size might not be reported accurately depending on the file system. For example, on the common *ext4* file system, an empty directory is 4 kB in size. This number remains constant until around 340 entries, when it grows to 12 kB, and then further in 4 kB increments for further hundreds of entries. The reported size of files corresponds to the volume of data stored in them, although files with explicit sparse data (i.e. with gaps in between significant data portions) may occupy less space on the storage medium.

The permissions are represented as a triple referring to the assigned owner of the file, users of the assigned group, and all other system users. Each set of permissions in turn consists of the read permission (`r`), write permission (`r`) and execute/search permission (`x`). The latter one refers to the permission to execute a file or to search through a directory and may be replaced by `t` for directories, which implies searching but with the additional constraint that only owners of the file in that directory (and the super-user) may delete it. The directory for storing temporary information (`/tmp`) is a case of `rwt`. Permissions can also be represented numerically as sums of permission bits, with the formula `x=1,w=2,r=4`. Hence, a permission of `755` (owner: `rwx`, group/others: `rx`) refers to normal directories and `644` (owner: `rw`, group/others: `r`) to normal files.

Typing `pwd` reveals the current working directory of the shell, thus providing an entry point to the file system. The output should be consistent with the environment variable `$PWD`. To find out the effective location, even in the presence of symbolic links, `pwd -P` can be used with confidence. Directly after login, the user's home directory is set as working directory. After logging in, the user can traverse the file system and therefore the working directory can be changed anytime with the command `cd <dir>` (change directory). The command without arguments (just `cd`) resets the working directory to the home directory. Special shortcuts are `cd ..`, changing to the parent directory, and `cd -`, changing back to the previous directory. A user can only change into directories with appropriate `rx` permissions. Moreover, shell sessions might be

restricted to prohibit directory traversal at all, especially in case the shell is launched through `bash -r`, although this is rarely seen in practice for real system users. Such facilities are rather used to lock down user accounts assigned to automatic program execution.

With the `ls` command, all files and directories within the current working directory are displayed along with their most important metadata, such as the last modification date. Common invocations include the options `-l` for the long view with all properties and `-a` to show all files including hidden ones, i.e. starting with a dot like `.hidden`. The physical occupation of blocks on the storage medium is shown with `-s`, and the reported sizes can be made human-friendly with `-h` similar to the same option in `df`/`free`. The output can be furthermore sorted by size (`-S`) or by modification date (`-t`).

With `stat`, all relevant metadata of files and directories such as size and timestamps are summarised. To get a tree-like view or calculate the disk usage of all subdirectories and files contained within, the `tree` and `du` commands complement its functionality. The latter command is typically invoked as `du -sh` to summarise disk usage in human-readable form.

Files and directories can be copied with the `cp [-r]` command and removed with the `rm [-r][-f]` commands, requiring recursive operation for directories and optionally forced mode to avoid confirmation prompts in batch scripts. The behaviour of the copy command differs depending on whether the target is an existing directory or not. For instance, `cp file1 dir/` assuming the directory exists copies the file into the directory under its original name (i.e. `dir/file1`), whereas `cp file1 file2` create a copy of the file under a different name in the same directory. If multiple source files are specified, the target is required to be a directory in any case (`cp file1 file2 dir/`).

While `cp` duplicates the content of files and directories on the storage medium, symbolic links (*symlinks*) can be used to create only references without requiring additional space apart from a single file system entry. The canonical invocation is `ln -s <source> <target>`, with source being an absolute path or relative to the target. The output of `ls -l` then shows an arrow in the form of `<target> -> <source>`. Apart from that, the link can be used normally just like the source file or directory, and removing it does not affect the source.

Renaming and moving works with `mv` for files and directories, and with the regular expression-capable `rename` for larger collections. The same behavioural switch as with file copies applies. For instance, `mv file1 file2` renames a file, whereas `mv file1 dir/` moves the file into the specified directory assuming it exists. New directories would be created with `mkdir`. The trailing slash shown at the end of directory names can be omitted. They are shown to clarify the status of a name as directory, and it is considered good practice to include such slashes in own commands.

Working with files across computers can be facilitated over the secure copy (`scp`) command SSH as previously mentioned. It is typically invoked with a local and a remote component. For retrieving a remote file to the current working directory, the syntax is: `scp <user>@<server>:/path/to/file .` including the significant final dot. As the default path refers to the home directory, any remote file located in there can be specified without absolute path, as in: `scp <user>@server:fileinhomedir .` again with the final dot. For the inverse direction, pushing a local file into the remote location, the syntax is `scp localfile <user>@<server>:` with the significant final colon. Called with the flag `scp -r`, entire directories can be copied in either direction. If a directory is specified but the flag is forgotten, the command complains accordingly.

File attributes in terms of user, group and other permissions are modified with the `chmod` command, using either symbolic or numeric syntax. Typical examples include `chmod 755 <dir>` and `chmod u+rw,g+r,o+r <file>`. The default permissions for newly created files are typically set to the difference 022, meaning that write access from group members or other users is not possible, or whatever is configured with the tool `umask` per shell session. Advanced attributes such as undeletable or append-only, if supported by the underlying file system, can be set with `chattr` and listed with `lsattr`. Checksums of files can be created with the simple `sum`/`cksum` tools or with their more robust alternatives `md5sum` and `sha1sum`. These checksums can be used to determine whether the file content has changed including after a corrupted or interrupted transfer. Differences between two files can be produced with `diff` (contextual) as well as with `cmp` (byte for byte).

As outlined in the concepts, files can be compressed to save space on the storage medium or during transmission. Commands such as `unzip` or `tar xvf` are useful to extract their contents, while packing is achieved with `zip -r <*.zip> <files/folder>` for ZIP files and `tar czvf`/`tar cjvf` for archives with the common GZip and BZip2 compression schemes, respectively. The `a` flag selects the compression scheme automatically based on the file extensions, which is useful when switching schemes often, involving also the supported LZMA, LZIP, LZOP, XZ and ZStd. Uncompressing works similarly with the commands `unzip` and, again automating based on the file extension, `tar xvf`. Many applications can work with compressed archives transparently at least in read-only mode; this applies in particular to text viewers and editors.

Commands to repeat in alphabetic order: cd, chattr, chmod, cmp, cp, diff, du, ln, ls, lsattr, md5sum, mkdir, mv, pwd, rename, rm, scp, sha1sum, stat, tar, tree, zip/unzip

Environment variables to repeat: $PWD

5.4.5 Creating, viewing and editing files

The `cat` and `tac` commands output the contents of a text file or multiple files (concatenated, hence the name) in forward and backward order to the standard output. The command pair `head` and `tail` are also aimed at line-based formats and display only a portion at the beginning or the end, respectively. A useful option for `cat` is `-n` for numbering the output lines. One might argue that `tac` should have the same option symetrically, but while it does not, it is easy to emulate, for instance, with the command combination `tac <file> | cat -n`. The most common option for `head` and `tail` is `-`*N*, with *N* being the number of lines to reproduce, by default 10. This technique of dynamic options has grown historically and can also be expressed more according to conventions by `-n` *N*.

With binary files, all of these text-assuming output commands might mess up the screen unless proper options are given such as `cat -v`. Hence, the `less` command provides a more convenient view of portions of a file including a binary-safe mode, and `hexdump -C` gives a detailed account of binary contents on the byte level. All file-related commands take one or multiple filenames as command-line argument, in addition to further options, as in `cat -n <filename>` (short for `--number`).

The `dd` command is useful for working with binary files. It allows for creating files with null content or random content as well as partial copies, such as: `dd if=/dev/zero of=/tmp/mynull bs=1 count=100`. For that tool, the options do not take leading dashes. They refer to the input file, output file, blocksize and number of blocks, respectively. The special device files usable as data sources are `zero` and `random`/`urandom`, and the one usable as blackhole data sink is `null`. Custom FIFOs (first in first out queues) as temporary buffers allowing for both writing and once reading the same content may be created with `mkfifo`. Files can be created by output redirection, such as `echo "text" > <file>`. Empty regular files can also be created with the `touch` command.

For interactive text file content modification, several text-mode editors exist. While working with them might not be easy in the beginning for people not used to text-mode tools, they are inherently powerful and can be found ubiquitously across systems so that it pays off learning them, especially as an engineer, to avoid time-costly roundtrips between a local editor and file copy commands. On the other hand, the file synchronisation might be automated, avoiding the need for editing in such a way. In practice, constraints differ and thus a basic mastering of these editors is still recommended. Common editors are `vi` (and its modern cousin `vim`[5]) as well as `nano` and `joe`. If the choice of editor does not matter, the `editor` command always works and brings up one that is installed. On the other hand, this alias is sometimes

[5]Vim website: `https://www.vim.org/`

invoked automatically, and it might be necessary to understand at least the basic commands in these three editors to not be surprised by the appearance while being completely lost in them.

With `nano`, the user gets to see a title line containing the name of the editor, and two footer lines containing hints for the most important keyboard combinations. Specifically, Ctrl+x quits (and asks whether changes should be saved if any), whereas Ctrl+o saves changes without quitting. With `joe`, or Joe's Own Editor, as it announced in the footer line on startup, the corresponding combinations are Ctrl+k;q for quitting with asking for saving changes, Ctrl+k;x for quitting with unconditional saving, Ctrl+k;d for saving without quitting, and Ctrl+c for an attempted quit without saving with asking in case anything has changed.

In `vim <filename>`, there is a concept of editor states (modes). As a result, text editing works within bounded sessions. An editing session starts by pressing the i key, and the escape key Esc ends it again. Direct character-by-character text modification is only possible within the editing session, whereas control commands work in between sessions, including after startup of the editor. These control commands allow for rule-based editing especially of larger portions of text.

Among the often-used control commands are copy (v/V) and paste (p/P). Similarly, this applied to navigation commands: gg jumps to the top, G to the bottom. Pressing a number followed by the arrow key jumps in the indicated direction by that number of rows or lines, for instance, 3↑ would place the cursor three lines above the current line. Numbers can also appear in other contexts as multipliers. Search works by pressing the slash key / followed by the search term (that can be a regular expression) followed by the Enter key. Working outside of the editing session also applies to further modification commands beyond paste. The command x deletes one character; multiple characters can be deleted by indicating the distance with a number and the direction with the cursor (d;number;←), and dd cuts a whole line. An often-used advanced command is the visual block selection (Ctrl+v and pressing cursor up/down ↑↓) along with insertion of characters at the beginning of the block (I; characters; Esc). This can be used to comment out a whole block of lines with the comment characters defined in a programming language or data format (`#`, `//`, `%` among others). For more rule-based editing, custom regular expression may be applied, for instance, with `:%s/before/after/`.

A whole range of such commands starting with colon (:) are available, including the program execution with `:!<program with arguments>`, which temporarily pauses the editor until the command returns. Entire files can be pasted with `:r <filename>`, and as the logical combination of the two commands, `:r! <program with arguments>`, executes a program and pastes its output into the currently edited file. For more productive editing with mul-

tiple views on the same file or different files, there is also `:split`/`:vsplit` to create new panes between which the user can navigate with the sequence Ctrl+w;w. The command without parameters creates another view on the same file, whereas `:(v)split <filename>` directly loads another file for concurrent editing.

The colon commands also govern how to save changes and leave the Vim editor, similar to what was explained for Nano and Joe. A `:w` command saves all edits to a file. A new file can be created by starting `vim` without arguments and the command `:w <filename>`. The command `:x` unconditionally saves and quits, `:q` quits a pane the editor if being in the last pane but asks if changes need to be saved, and `:q!` unconditionally quits without saving, potentially losing changes.

Encodings are an important concern when working with data files, especially those downloaded from various Internet sources. The canonical encoding should be Unicode, with its standard representation UTF-8. Sometimes, legacy encodings appear and are either handled on a case by case basis by specifying the corresponding encoding at each file open operation in Python, or the file is converted before use. With Vim, differently encoded files are shown as `[converted]` upon startup, and details about the file encoding can be shown with `:set` watching for the `fileencoding` setting. The appropriate command to fix the encoding is `:set fileencoding=utf-8` followed by a save `:w`.

> Commands to repeat in alphabetic order: cat, dd, editor, head, hexdump, joe, less, mkfifo, nano, tac, tail, touch, vi/vim

5.4.6 Networking

A number of commands are very useful to know and to master in networked environments beyond SSH and ping: `netstat`/`ss` and `netcat` for basic networking interaction, and `wget`/`curl`[6] to fetch files from the web and interact with web services.

An invocation of `netstat -ltp` shows all port numbers occupied by TCP-listening services, including the service names of those that run under the same user account. The parameters relate to showing listening services (`-l`), restricting to TCP connections (`-t`) and showing the program names (`-p`). Program names are only shown for accessible processes, i.e. own processes of the calling user or all processes when the tool is invoked by the super user (root). Even without this option, the PID is shown, allowing stray processes to be terminated to liberate port numbers. Depending on the service implementation, the port may still be occupied for a number of seconds before it is finally liberated. Another useful piece of information is the bind address.

[6] Curl: `https://curl.se/`

If it says `::1` or `localhost`, the service is only accessible from localhost; or otherwise, from outside as well. Bind hosts are resolved unless numeric display (`-n`) is requested.

The command `netcat` can be used to test simple client/server connections. First, `netcat -l -p <port>` starts listening on a certain port number. A `netstat` on a second terminal can verify that the service works. Then, `netcat <host> <port>` connects to this service, allowing for bidirectional communication. The host can be set to `localhost` if running on the same host as the service, and it can also be set to an IP address such as `127.0.0.1`. Of course, the client can also be used to communicate with existing services, as long as they accept a text-based protocol, by specifying a different hostname and port number.

With `wget <url>` and `curl <url>`, it is possible to download files from HTTP servers. The Wget command by default saves the file, whereas Curl pastes its contents on the terminal, making it suitable for rather small files unless additional parameters are set. Additionally, Curl makes it trivial to interact with web services, including the ability to post data. For instance, the command `curl -X POST "https://httpbin.org/post?a=b" -H "accept: application/json"` submits a JSON request to a web service that parses the URL arguments and returns an informational JSON structure about them and other arguments not used in this example. Some web services also require the content type to be set in order to recognise posted content. Setting another header solves this problem, as in `-H "content-type: text/csv"`.

For security reasons, it should be pointed out that the often-seen combination of curling and executing a script from untrusted sources (e.g. `curl https://... | sudo sh`) on trusted systems is discouraged. Such commands should be decomposed, executed manually and complemented with a brief validation of the file contents. This also applies to any advice given on the Internet on running certain scripts and commands.

On the other hand, too tight security often gets in the way of learning commands. In case a certificate cannot be validated over an HTTPS connection, both commands allow for switching off the check, at the expense of reduced security: `wget --no-check-certificate` and `curl --insecure` (alias `curl -k`).

Commands to repeat in alphabetic order: curl, netcat, netstat, ss, wget

5.4.7 System administration

Administering a system means being responsible for it. The system in question can be a data scientist's workstation or notebook, a physical server hosted at the company premises or a dedicated data centre, a virtual machine obtained

from a cloud provider offering computing infrastructure as a service or a container running on any of those. Typical administration tasks include regular housecleaning tasks, i.e. removal of cruft such as old files and stray processes to reduce the resource utilisation, keeping the system up to date and secure, the installation of additional software needed for being productive, and ensuring automation and auditability. Sometimes, unplanned maintenance needs to be performed, for instance, restarting a service that crashed, evidently after having found out about this situation.

After having obtained information about the system resources, the first step on any system is usually awareness about what flavour of the operating system is running. This knowledge is helpful when consulting documentation on any problem or when looking for software packages to install. If the operating system is self-installed, this knowledge is usually given, but otherwise it needs to be obtained. Due to many evolving flavours, there is no systematic way to learn the differences in operation and administration. Rather, using different systems once in a while helps in building up experience.

On Linux systems, there is also no systematic way but rather a heuristic approach. If the configuration file `/etc/apt/sources.list` exists, it hints at the presence of a Debian-based system, e.g. plain Debian[7] (indicated by the existence of `/etc/debian_version`) or Debian derivatives such as Ubuntu. These flavours or more specifically these distributions have emerged as most popular ones over time. They have existed since the mid-1990s and since the mid-2000s, respectively, and are largely maintained by volunteers around the world, with plenty of forums and mailing lists available for help. If the file `/etc/lsb-release` exists, it may give further information about the operating system distribution. Other popular distributions include Arch Linux, indicated by `/etc/pacman.conf`, Red Hat/CentOS and SUSE. Once a flavour or distribution is known, it can be put to work.

Providing services on a computer or performing complex data analysis tasks inevitably leads to long-running processes. Unless a computer is rebooted or the process crashes, the execution time of a long-running process may be in the order of hours, days or even months.

Long-running processes ideally provide their own means for inspection, such as status and progress messages, log files or dashboards. In absence of these means, to find out what a process is doing, the generic `strace` (system call trace) wrapper command provides insights into running processes on the operating system call (*syscall*) level. Although not all activities - such as complex numerical calculations and other internal logic - show up this way, it is possible to understand much of the OS-interfacing behaviour of the processes monitored with the command prepended to the usual command name and arguments. For instance, `strace ls -l >/dev/null` reveals how the list of files and subdirec-

[7]Debian universal operating system website: `https://www.debian.org/`

tories is read from the current working directory, potentially following child processes (`-f`) or attaching to a running process (`-p <pid>`). The wrapper also shows the wrapped command's writes to the terminal on standard output. The tracing information itself is written to the standard error channel, and due to output redirection, the tool's output is suppressed so that both do not mix and all terminal output comes from the tracing. Somewhere in the middle of much output, the experienced user can then spot an internal operating system call called `getdents64()`, the purpose of which is to get directory entries. These system calls are functions implemented in the programming language C within the Linux kernel. Even when programming in C close to the system, they are usually not invoked directly but rather through portable wrapper functions as part of the libc system library, in this case: `readdir()`, and those in turn are wrapped by the various Python modules close to the system, such as `os`, in this case by: `os.listdir()`. In the reverse direction, this allows for finding out what exactly a Python program is doing on the OS level.

Most operating systems do not have a concept of persistent sessions. All processes are lost when a reboot occurs. Moreover, when starting processes through SSH, they are often terminated along with the session, whereas it would be desired to keep them running automatically without having to invoke them through a screen wrapper. In case they crash, perhaps they should be even restarted automatically. Supervising and autostarting applications represent an important concept to achieve that and to ensure continuous service delivery. This applies to both user-level and system-level services. Technically, it involves setting up appropriate launch scripts and unit files.

With SystemD as supervisor, a system user first needs to be given the permissions by the super user to host long-running services with the command `sudo loginctl enable-linger <username>`. The user can then deploy and start a unit file with the unprivileged commands sequence: `cp my.service ~/.config/systemd/user` to register the unit file in the default location, followed by `systemctl --user enable my.service` to activate it upon next boot, and finally `systemctl --user start my.service` to also start it right away. Whether the service works correctly can be verified with the command `systemctl --user status my.service` and ultimately by checking the listening port with `netstat`. The service unit file content may be as follows, not taking automated restarts into account:

```
[Unit]
Description=My service
[Service]
ExecStart=python3 ...
WorkingDirectory=/home/username/mydir
[Install]
WantedBy=default.target
```

If the command is not a long-running service but rather something that should run in defined intervals, a Cron-like regular command invocation is possible by setting up an additional timer unit that references the sservice unit by name, either explicitly with the Unit specification, or implicitly by having the same name minus the suffix (i.e. `my.timer` in the example).

```
[Unit]
Description=My timer
[Timer]
OnUnitActiveSec=30 # or OnCalendar
AccuracySec=1
Unit=my.service
[Install]
WantedBy=timers.target
```

Several commands are related to the management of such system-wide services under the responsibility of the super user, including privileged services running as root. For all of those, the effective process user is set to root, expressed by the environment variable `$USER`. The command `sudo service --status-all` gives an overview about the registered services and indicates active ones with a + sign. A service (for instance an Apache web server listening on port 80 and thus required to be privileged) might be hanging and requires a restart. This requires knowing the service name and furthermore privileged execution through the `sudo` wrapper command, as follows: `sudo service apache restart`. The command informs the supervisor service to perform a lifecycle operation (start, stop, restart, reload configuration, status check) on an application service.

The same wrapper command Sudo can be used to edit global configuration files, for instance in the `/etc` folder that is write-protected to ordinary users, by carefully running `sudo editor <global-file>` or variations thereof. While incorrect use of an editor may always lead to a loss of data, privileged execution may furthermore lead to an unusable system, and therefore extra care needs to be applied to ensure the file is backed up beforehand. This can be achived with a simple `sudo cp <global-file> <global-file>.backup<date>` before editing, although more sophisticated version control and backup strategies exist and are explained later.

The way Sudo authenticates users is by asking for their system password. This makes it tricky to use Sudo in non-interactive scripts because the asking is interactive. As first precondition to gain privileged access, this password must be correct, but as second precondition, the user or a group the user is member of needs to be registered for being able to run Sudo in the file `/etc/sudoers.conf`.

Privileged command execution may also be necessary for other tasks. Among them is administering users, primarily the addition of new users and groups

with `adduser <username>` and `addgroup <groupname>` on Debian-based systems, and the similarly-named `useradd` and `groupadd` on other systems. For testing a new application, isolating the test by creating a custom user and group may be useful. To achieve the isolation, one would first create a new system account with `sudo adduser [--shell /bin/bash] [--home /home/testuser] --comment "" testuser`, with explicitly chosen login shell and home directory in case the default values are not applicable. This command first asks for the Sudo authentication, then twice for the new user's password, and with that information it creates the user, a corresponding group, and a home directory with default (skeleton) content. (It should be noted that the comment field may still appear as `gecos` on older systems, derived from human-readable information attached to the user account originally appearing in the General Comprehensive Operating System (GECOS).) The new user does not have Sudo privilege, and therefore an application run under that account can only cause minimal damage. To proceed, the interactive shell session switches to that new account (`sudo su - testuser`), testing can occur, and the test session can be closed with `exit`. Finally, the custom user can be removed along with the generated group and all files with `sudo deluser --remove-home testuser`.

Privileged commands are also required to change modes and ownership of files owned by other system users (`sudo chgrp`, `sudo chown`) and to send messages to all local users with active shell sessions (`sudo wall`) on a shared login system. Executables can be configured to gain elevated privileges automatically upon execution with the set-user-id and set-group-id bits (SUID/SGID), as in: `chmod u+s <executable>`. Careful application design involves dropping these privileges as soon as the privilege-requiring operation has finished.

Lastly, system administration also involves checking log files occasionally. System-wide logs are stored in `/var/log`. Many of them are rotated as they can grow big, marked by a number after the log file name. Of primary interest is the file `syslog` that contains kernel and application messages. Appended lines to that file may be followed with `sudo tail -f /var/log/syslog`. The log directory also contains the information base for commands such as `last`, with binary files such as `lastlog` and `wtmp`, but primarily contains text files, sometimes in subdirectories for more complex applications. In addition, hardware-related issues might be found out with `sudo dmesg` (diagnostic messages).

Commands to repeat in alphabetic order: addgroup/groupadd, adduser/useradd, chown, chgrp, dmesg, loginctl, service, strace, sudo, systemctl, wall

Environment variables to repeat: $USER

Repetition

1. What is the difference between a *TERM* signal and a *KILL* signal?
2. When grepping for a list of processes, the `grep` command itself appears in the list. How can this be avoided?
3. How to count the number of current and past login sessions to a system?

5.5 Shell programming

The previously given information on shell variables, shell commands and useful tools is sufficient for occasional use of a system. For automation, a higher-level programming language such as Python may be used. Sometimes, though, data scientists are confronted with complex shell scripts containing all sorts of programming constructs. Bash in particular is therefore not only a command prompt but also a programming language on its own. Learning yet another language might not be favoured, but being able to understand this language in order to customise or extend scripts is nevertheless a useful skill. In this section, basic shell programming concepts and constructs are conveyed.

5.5.1 Vocabulary and interaction with scripts

Bash contains a number of built-in commands that, together with the executables on a system and user-defined functions, form its vocabulary. An overview can be obtained with `help` as well as more comprehensively (but missing some commands unfortunately) with `man builtins`. The vocabulary can be divided into flow control, job management, directory bookmarking, arguments parsing and other groups.

One characteristic behaviour of shell scripts is that, while syntax errors are fatal, execution errors in commands signalled with non-zero exit status are ignored by default. The offending lines are then simply skipped, and the errors in these commands are ignored. In many cases, a stricter behaviour closer to that of most programming languages is desired. Shell scripts therefore often start with the command `set -e`, which makes the shell exit immediately whenever a command returns a non-zero status.

In turn, this behaviour has led to the convention that tools as well as user-defined functions return a zero exit code upon successful termination, and a documented non-zero exit code otherwise. In Bash, `exit 1` would adhere to this convention. In Python, using `exit(1)` or, to include an error message, `exit("message")` achieves the intended behaviour. In other languages, similar constructs are available to signal a non-successful termination to the shell. The special variable `$?` reports about the exit status of synchronously executed child processes.

There are also conventions on how to parameterise tools and functions. Historically, four main flavours developed. These are short dashless parameters (`<program> x`), long dashless parameters (`<program> extract`), short dashed parameters (`<program> -x`) and, finally, more self-explaining but requiring more typing effort, also long double-dashed parameters (`<program> --extract`). The first flavour is still in use with some historic tools but has largely fallen out of favour, such as `ar x <archive.ar>` for uncompressing a compressed file. The second flavour is typically used for subcommands offered by a complex command, such as `git clone`. The third and fourth flavour are very common and often go hand in hand as equivalents, such as `vim -h` equal to `vim --help`. Short options can also be combined, such that `-x -y` is equivalent to `-xy`. Due to the limited set of characters in the Latin alphabet that are traditionally used for the third flavour, complex programs only assign such short parameters to the most important options. Sometimes, they match with the first letter with the long parameter but sometimes they do not, something to be aware of. For those last two flavours, there are parameters with and without argument values (e.g. `-s` or `--strict`, either binary or with a default value, versus `-s high` or `--strictness high`). The argument value may often also be appended with equal sign, as in `--strictness=high`, and for short-style parameters also without space (`-shigh`). Its type (string, numeral, boolean, filename or URL) is defined by the application. In addition, commands may take a variable number of arguments, often files or URLs, given at the end of the invocation line. Hence, it is not unusual to see a command with mixed parameters and arguments in the following form: `prog -xy --strict a.py b.py`. In this book, for brevity and rapid practice, the short-style parameters are used in most places, sometimes complemented by their longer equivalents.

Shell scripts can thus be parameterised with arguments. Each parameter is given a numerical variable, with `$0` referring to the script itself and `$1` to the first parameter. If the parameter is not supplied, the variable remains unset and empty. The special variables `$@` and `$*` refer to the whole argument vector containing all parameters as space-separated list. The difference is in further passing the arguments internally to functions or commands. The variant `"$*"` joins the arguments into one (i.e. becomes `$1` in the called function or command), whereas `"$@"` keeps the list intact and passes each argument separately.

More sophisticated parameter parsing is possible with the `getopts` instruction. This command is called repetitively on an arguments list and is able to process short-style parameters with and without argument values. For instance, the following script can be called as `script -a -b -c x` or in various combinations thereof and allows its subsequent logic to be adapted to the parameters. The variable `$var` contains one of the valid parameters in each

iteration, and `$OPTARG` the argument value if indicated with a colon as is the case for c (`c:`). The loop and condition contained in this script are explained in detail further below.

```
while true
do
  getopts "abc:" var "$@"

  if [ "$var" = "?" ]
  then
    break
  fi
  echo "** $var [$OPTARG]"
done
```

Testing for empty or unset variables can be done with `test -z "$1"` or alternatively `[ -z "$1" ]`, a condition that returns 0 if the variable is empty, and 1 otherwise.

Text including variables is output by `echo`. The counterpart to assign variables with user-defined input is `read`. A typical invocation is by including a prompt to tell the user what the input should be, as in `read -p "Prompt? " var`. The variable `$var` can then be used for further processing.

Dynamic command evaluation can be performed with `eval`, a command to be used sparsely and with extreme care concerning user input. For example, `eval "ls -l"` interprets the provided string argument as a shell command to execute, and `a=9; eval "sleep $a"` builds and runs a variable-dependent command. Many more built-in commands exist, but three groups are now explained in greater detail: job management, control flow programming, and function definition.

> Commands to repeat in alphabetic order: eval (built-in), getopts (built-in), read (built-in), set (built-in), test (built-in)
>
> Environment variables to repeat: $*, $@, $0..$n$

5.5.2 Job management

Jobs in the context of a shell refers to active processes on the OS level that have been spawned from that shell. In addition to global process management tools (`pidof`, `kill` etc.), the shell offers additional management functionality for the commands directly under its control.

The command `jobs` displays all jobs. For instance, running `sleep 10 &` in the background immediately followed by `jobs -l` shows the still running sleep command along with its PID and its shell-internal job number in long

format. In case the process terminates, it appears one more time with the status finished before disappearing from that list. Any backgrounded job can be foregrounded, taking over the terminal, with `fg <job>`. A command run in the foreground can be suspended by pressing Ctrl+Z, assigning it a job number. The job can then continue in the background with the command `bg <job>`. A foreground job can also be terminated with Ctrl+C. Hence, any job is in the status of running, suspended, or finished.

The running shell itself may be terminated with `exit` or suspended with `suspend`. The suspension blocks the shell entirely unless it receives a continuation signal (`kill -CONT <pid>|<job>`). With many active shells, finding out the PID of a just suspended shell is not trivial but becomes easier knowing that the output of `ps` shows the status `Ts+`. In a running shell, the special variable `$$` informs about the PID. Shell suspension is rarely needed in practice. It should be noted that the shell command `kill`, similar to `time` and `echo`, shadows the identically named executable and is therefore able to control both processes and jobs under the control of the shell.

The command `history` shows all shell commands executed in the current session as well as in previous sessions based on session recording.

> Commands to repeat in alphabetic order: bg (built-in), fg (built-in), history (built-in), jobs (built-in), kill (built-in & executable), suspend (built-in)
>
> Environment variables to repeat: $$

5.5.3 Control flow programming

Bash is an imperative programming language based on conditional branching and looping. Such constructs can be written across multiple lines but can also be shortcut by using the semicolon (`;`) in place of line breaks. *While* loops follow the form `while <condition>; do <command>; done`. *For* loops iterate over a sequence of values or a globbed list of files, as follows: `for <var> in <list>; do <command>; done`. Loops can be controlled with `continue`, jumping back to the first line of the loop body, and `break`, jumping out of the loop.

Sequences for iteration are parsed as strings divided by a separator, the internal field separator (`IFS`), which by default is set to the space character. This may collide with spaces in file names, not when iterating over files directly but for instance when reading a list of files from another file. Hence, the IFS can temporarily be set to something else which is unlikely to occur as character in filenames, such as line breaks. The following code achieves that: `IFS=$'\n'; for fn in ‘cat myfileslist‘; do echo "* $fn!"; done`. However, this

alters the field separator beyond the end of the command, so that it should be saved and restored (`ORIGIFS=$IFS; ...; IFS=$ORIGIFS`).

Branching and conditional execution may be based on `if` or `case`. The canonical form of a choice between two branches is then either `if <condition>; then <command>; else <command>; done` or `case <var> in <pattern1>) <command1>;; <pattern2>) <command2>;; esac`.

> Commands to repeat in alphabetic order: break (built-in), case/esac (built-in), continue (built-in), for/do/done (built-in), if/then/fi (built-in), while/do/done (built-in)
>
> Environment variables to repeat: $IFS

5.5.4 Shell functions definition

Functions are declared by simply typing the function name followed by a pair of parentheses, and another pair of curly braces containing commands with significant spaces in between. Similar to a shell script, a function may receive arguments, which are numbered starting from 1. Hence, the function declaration `func(){ echo "$1"; }` leads to a new command `func <argument>` in the running shell.

Functions are either invoked explicitly or based on events such as OS signals. The `trap` command can be used to intercept system signals such as SIGTERM and SIGCONT in addition to the two user-defined signals SIGUSR1 and SIGUSR2. For instance, `trap func USR1` runs the previously defined function whenever a signal is received, which can be verified with `kill -USR1 $$`. All active traps can be printed with `trap -p`. In practice, trapping is in demand for cleanup actions that need to be performed no matter whether a script is terminated regularly or irregularly.

> Commands to repeat in alphabetic order: kill (built-in), trap (built-in)

Repetition

1. What may happen if a user types `xyz` into a shell?
2. What happens when the command `kill $$` is invoked?

5.6 Python modules for OS interaction

Although Bash programming is great for automation at the operating system level, it may be in itself tedious especially for higher-level applications that

require structure, modularity and more powerful built-in functions. For those, writing the corresponding code in Python, invoking Python scripts from the shell or combining Python code with OS-level tools might be the better application design. Hence, in the following, six additional Python modules (out of the larger built-in module list[8]) of high relevance in the interaction between Python code and the OS are introduced, and complementary information of interacting with Python on the shell is given.

5.6.1 Running the Python interpreter

The default Python interpreter is cPython. It is implemented in the C programming language and can be invoked on the command line by its versioned interpreter name `python3` or, as used throughout this book, its alias name `python`. As the Python programming language evolves, checking the version with `python -V` is advisable. For pragmatic reasons, the book assumes the version to be at least 3.10, although previous versions 3.x should also work fine for most of the examples.

The Python interpreter enters interactive mode by default similar to Bash. More specifically, it follows the Read-Evaluate-Print-Loop (REPL) paradigm. First, it reads user input with the prompt indicating a new command (`>>>`) or a continuation of a previous line (`...`). Then, it evaluates the input in the form of an expression or a statement, such as variable assignments, mathematical formulas, functions, control flow instructions. Afterwards, it prints out the return value of any statement or expression, unless it is `None`, before closing the loop and reading the next input. The interpreter can be quit with the command `exit()` or with the keyboard combination Ctrl+d.

When invoked in the form of `python <scriptname.py>`, the interpreter instead executes the specified script in batch mode. Nothing is then printed unless the `print()` function is used or an uncaught exception is raised, and no interaction happens unless the `input()` function is used. A one-liner script consisting of one statement or expression or a sequence of those separated by semicolon can also be passed directly on the command line with `python -c "<command>"`. Similarly, `python -m <module>` loads a module and executes it as script, i.e. typically the guarded main code.

There are several environment variables and options that influence the behaviour of the Python interpreter. Not all of them can be documented here, but among the more often used ones are running `python -u` for unbuffered output especially when piping the output to another tool invoked on the shell; and informing about additional module search paths beyond the default (informed as `sys.path`) with the environment variable `PYTHONPATH`.

Python scripts can be made executable by using `chmod +x` and setting

[8]Python modules: `https://docs.python.org/3/library/index.html`

`#!/usr/bin/env python` as their shebang line. Due to Python being an interpreted language, similar to Bash, all Python scripts appear in the OS process list as merely `python`. The script name appears at least as second parameter so that processes can be distinguished with `ps x`, but using process management tools such as `pidof` or `killall` is then challenging. An external Python module `setproctitle` can be installed from PyPI (`pip install setproctitle`) or from the distribution repository (e.g. on Debian-based systems with `sudo apt-get install python3-setproctitle`) as solution. At the start of a Python script, calling `setproctitle.setproctitle("title")` then allows for giving a unique name.

In general, it is important to consider the effects of Python code on performance and memory use especially when handling very large volumes of data. Performance may slightly degrade especially for multi-threaded and iterative operations, which can be remedied by choosing the right interpreter and data-processing modules. Compared to the standard cPython interpreter, PyPy is optimised for higher performance and better threading support and can be invoked as `pypy3` to give a faster and more scalable drop-in replacement. Further alternative interpreters exist, notably iPython, which is known for its integration into Jupyter notebooks, and MyPy as an emerging interpreter with support for static typing enforcement. Memory usage in the interpreter may also increase due to data being represented with metadata in memory. Using libraries such as Numpy and Pandas that manage raw data without the metadata helps to avoid this issue.

Commands to repeat in alphabetic order: python, pypy3

Module functions: setproctitle.setproctitle (external)

5.6.2 Modules 'os' and 'sys'

The `os` module, along with its submodule `os.path`, represents the main interface to low-level operating system functionality.[9] It wraps many of the functions contained in the main system libraries to make them accessible from Python in a portable manner. Achieving portability requires discipline in using the module's OS abstractions without compromises. For this matter, differences in operating systems are masked, for example, path separators (`/` versus `\` as `os.pathsep`). On Linux, the `os` module is primarily the interface to `libc`, which in turn provides convenience functions to access low-level functions in the kernel itself. Therefore, calling one of this functions switches the execution context from user space to kernel space temporarily. This is not different

[9] OS module documentation: `https://docs.python.org/3/library/os.html`

from `print`, which is, however, used so often that it was turned into a Python built-in function instead of being exported through this module.

Among the commonly used functions is `os.system` to spawn a new process. On Linux, the command is interpreted as a shell command, so that all shell facilities such as pipes and redirections are available. An example invocation is `os.system("ls -l > /tmp/listing")`. Proper care must be applied to avoid using untrusted input to that function and for properly quoting the command. Other useful commands are `os.cpu_count` to get information about the possible parallelism in multiprocessing and `os.kill` to send signals to processes.

To interact with environment variables, the functions `os.getenv` retrieves values of variables passed to the Python interpreter with default values for unset variables, for instance, `os.getenv("LANG", "C")`. Its counterpart is `os.putenv` to update the values or introduce new environment variables. It should be noted that this only affects processes spawned from the current one. To effectively update variables within the running Python session itself, the dictionary-like data structure holding the environment must be modified directly, following the form `os.environ["LANG"] = "C"`. To avoid a spoofed user identity, `os.getlogin` is preferred over reading out the value of `$USER`, albeit it does not follow Sudo semantics.

Among the often-used functions related to directory trees are `os.getcwd` to report the current working directory, `os.chdir` to change to a different directory, `os.makedirs` to create a new directory unless it already exists, and `os.listdir` to report the directory contents. Helpful functions from the submodule are `os.path.join` to construct subpaths for the navigation, the reverse with `os.path.basename` to split off the rightmost file or directory name and `os.path.dirname` for the remainder, and `os.path.isdir`/`os.path.isfile` to check the type of entry when iterating over a directory as reported by `os.listdir`.

The `os` module is complemented by `sys` for some OS interaction, mostly related to the process itself.[10] This encompasses the handover of parameters to the Python interpreter or script (`sys.argv`), the search paths for executables to be spawned (`sys.path`) and the default communication channels for input, output and errors, respectively (`sys.stdin`, `sys.stdout`, `sys.stderr`).

To request the termination of a Python process, even multiple functions are available: The built-in functions `exit` and `quit` take either a numerical exit code argument (0 meaning success, otherwise failure) or a text message (`None` meaning success, otherwise failure message). Failures are treated as exit code 1, represented by a raised `SystemExit` exception, with appropriate error message, if specified, written on the standard error channel. These methods can be overridden in interactive sessions and are unavailable in some

[10] Sys module documentation: `https://docs.python.org/3/library/sys.html`

interpreters like iPython when running scripts. Therefore, it is advisable to use `sys.exit`, which takes the same parameters. Still, the exception may be caught by higher-level code, which is often the desired behaviour but as a consequence does not guarantee termination. Alternative options for forced termination are `os._exit`, taking only a numerical exit code, and `os.abort`, immediately terminating the process with a failure code.

> Module functions: exit (built-in), quit (built-in); os.abort, os.chdir, os.cpu_count, os.getcwd, os.getenv, os.kill, os.listdir, os.makedirs, os.putenv, os.system; os.path.basename, os.path.isfile, os.path.isdir, os.path.join; sys.exit

5.6.3 Module 'shutil'

This module[11] refers to *shell utilities* and represents the equivalent of many file and directory management commands typically invoked on the OS shell. These include commands to copy or move files (`shutil.copy`, `shutil.copytree`, `shutil.move`), to change their ownership (`shutil.chown`) or to get the cumulative disk usage of a directory path (`shutil.disk_usage`). To copy a single file, for instance, the syntax would be `shutil.copy("origfile.txt", "filecopy.txt")`. Like all I/O operations, it should be enclosed in a try-except block to check for typical errors such as disk full or insufficient permissions. Recursion is used by `shutil.copytree` and `shutil.move` if the second parameter is a target directory so that entire directory hierarchies can be copied or moved. The module can also be used to pack and unpack an archive such as a ZIP file or compressed TAR files (`shutil.make_archive`, `shutil.unpack_archive`). This resembles the invocation of the compression utilities in the shell, whereas for more fine-grained support, dedicated Python modules for compression and archiving are available.[12]

> Module functions: shutil.chown, shutil.disk_usage, shutil.copy, shutil.copytree, shutil.make_archive/ shutil.unpack_archive, shutil.move

5.6.4 Module 'tempfile'

The module enables the safe creation of temporary files and directories with unique names in a shared directory, by default the standard directory for temporary files, which is usually at risk for race conditions when multiple applications would create a file with the same name. On Unix-like systems, the directory for temporary files is `/tmp`, whereas on Windows, they can be found in

[11] Shutil module documentation: `https://docs.python.org/3/library/shutil.html`

[12] Python modules for compression: `https://docs.python.org/3/library/archiving.html`

`C:\Windows\Temp` and `C:\Users\<user>\AppData\Local\Temp`. The canonical call for the safe opening of temporary files for write access is to the constructor of the class `NamedTemporaryFile` with appropriate parameters. For example, in a web service to handle image uploads, the following code would ensure that all write access is redirected to an unprivileged area first: `f = tempfile.NamedTemporaryFile(suffix=".jpg"); print(f.name)`, followed by `f.write(b"..."); f.close()`. As long as the file is not closed, the attribute `name` of the object can be used to share the file content with other processes.

Module classes: tempfile.NamedTemporaryFile

5.6.5 Module 'argparse'

To facilitate the interface between shell and application, this module provides a structured way to access command-line parameters that influence the actions of the application. It is based on the convention that an application can be invoked with a binary parameter or flag (e.g. `app --strict`), a qualified parameter with argument value (`app --strictness 5`), subcommands (`app check ...`) and an arbitrary number of arguments (`app 1.txt 2.txt 3.txt`). The module provides the `ArgumentParser` class that is first constructed with information about the permissible flags and arguments. The information encompasses even typing information to inform that an argument takes only numeric values or references to files. Then, the parser object is applied on the standard Python interface for command line arguments, i.e. `sys.argv` excluding the script name itself, and sets attributes on the returned arguments object that the application can evaluate. A minimal use case would thus be as shown in the listing below.

```
import sys
import argparse
p = argparse.ArgumentParser()
p.add_argument("--url", default=None, help="URL to load")
args = p.parse_args(sys.argv[1:])
print(args.url)
```

Documentation about the supported invocation options is automatically created from the parser object and made available via the long-style `--help` parameter or its short-style equivalent `-h`. Moreover, in case incorrect options are supplied, an error message with a short synopsis of correct options is generated automatically, written to the standard error channel and causing a non-zero exit status. The module is therefore the first step in input validation, although further checks on the validity of parameters and arguments should be performed within the application code.

Module classes: argparse.ArgumentParser

5.6.6 Module 'subprocess'

Python processes can spawn synchronously executing (i.e. blocking) subprocesses via `os.system("command ...")`. This function returns the exit code and therefore lets the calling application code decide on the convention of zero meaning success and non-zero meaning failure how to proceed. However, this function is unable to return any output from the executed command. The output is instead just written to the standard output channel of the terminal. Moreover, it does not easily permit the execution in the background, instead requiring the use of multithreading for this purpose or appending `&` to the command but then returning immediately without being able to track when the command finishes. Hence, the `subprocess` module is a more versatile alternative, giving more control about subprocesses handling to the application engineer, in return for a slightly more complicated invocation interface. The canonical invocation behaviourally equivalent to `os.system` is as follows: `p = subprocess.run("command ...", shell=True, stdout=subprocess.PIPE)`. The return value is an object encapsulating information about the spawned process. This can be followed by `p.wait()` to wait for the command to finish, `print(p.returncode)` to inform about the success (0 meaning success by convention), and `print(p.stdout.read().decode())` to access the command's standard output.

Module functions: subprocess.run

5.6.7 Module 'socket'

The socket module allows re-implementing the Netcat command in Python and adding application-specific logic on top of it. On the server side, it permits the creation of a socket by calling `s = socket.socket()` and binding it to a network interface specified by IP address and port number, as follows: `s.bind(("0.0.0.0", 9999))`. The special IP address, equivalent to the empty string `""`, instructs the OS to bind to all interfaces. In practical terms, the service is then accessible from incoming connections originating outside of the computer. Alternatively, the address `"127.0.0.1"` can be used to allow only local connections. Once the socket is created, it can be used to listen to incoming connections (`s.listen(1)`) and to represent the client-specific socket along with client address information after a successful connection (`c, caddr = s.accept()`). The client object then offers basic network interaction with the `recv`, `send` and, eventually, `close` methods. Sending and receiving data works similar to reading and writing binary files. On the client side, the

interface is similar but omits the creation of additional sockets. A socket offers the `connect` method that mirrors `bind`, and then allows for sending and receiving directly on that socket.

While these socket methods mirror the low-level system functions, a slightly more comfortable interface is available. On the server side, the function `socket.create_server` creates a socket ready to accept without requiring the binding and listening steps; on the client side, the corresponding function `socket.create_connection` sets up the connection with control over timeouts.

The historic OS behaviour is that, whenever a program terminates, any port that it bound to as a server remains occupied for a short period of time. Restarting then usually fails unless precautions were taken to tell the OS to disable this reservation. The higher-level interface makes these easy to integrate, as in: `socket.create_server(("", 2000), reuse_port=True)`.

To access HTTP services, the Python equivalent of the Curl/Wget functionality can be achieved with the default module `urllib.request`, as well as the third-party module `requests`. Both provide extensive support for HTTP GET and POST requests, but are not explained in detail here.

Module functions: socket.create_connection, socket.create_server, socket.socket

Repetition

1. What would the command `python -c 'import os; print(os.getpid())'` do?
2. How to programmatically create a directory in an idempotent way, i.e. adding it if it does not already exist, and do nothing otherwise?

5.7 Package management

Customising and maintaining an operating system requires understanding the lifecycle of software and data, their representation as layered and strongly related packages and the use of package managers to achieve a desired system state. While essential functionality and many basic tools might be already pre-installed, several complex data-processing and analytics packages for data scientists require not only a one-time additional installation, but even tracking new releases systematically and upgrading the workstation or server accordingly. One advantage of using provisioned packages through appropriate package managers is that all necessary dependencies are automatically installed. This reduces the cognitive load and allows focussing on what functionality

should be present, without having to understand how that functionality is implemented. A second advantage is that, instead of having to trust dozens or hundreds of different download locations, many package repositories take at least basic precautions to avoid low-quality and malicious packages.

Packages emerge within certain ecosystems that are developed by communities. Some communities work close-knit almost like teams, while others are rather loose sets of people. Ecosystems are also either thematically focused on operating systems, languages or technologies, or they are rather universal. Navigating these ecosystems is not trivial. In this book, the emphasis is on Python packages, OS-level packages, and (in the subsequent section) Docker images. Understanding those concerning lifecycle, tool support and obstacles also opens the door to understanding others.

5.7.1 Python package management with Pip

When it comes to extending the functionality of the installed Python interpreter, the language-specific ecosystem of packages provides a huge diversity and generally a high quality of popular packages. The `pip` tool is the package installer for Python and the main interface to this ecosystem. Before the installation, the scope needs to be defined. A package can be installed system-wide by invoking Pip as super-user, or per user by invoking Pip as that user, or even confined to a single project's virtual environment in conjunction with additional tools such as `virtualenv`[13] and the derived `venv` library. To avoid clashes with packages already provided by the operating system, the use of `virtualenv` or `venv` can even be mandatory.[14]

On most Linux installations, the default directory for system-wide Python package installation from third-pary sources is in the `/usr/local` hierarchy, specifically `/usr/local/lib/python3.`X`/dist-packages`. Correspondingly, for per-user packages it is `~/.local/lib/python3.`X`/site-packages`, with X referring to the minor version (e.g. 10 for Python 3.10). The OS distribution itself may have placed its curated packages into the `/usr` hierarchy, specifically `/usr/lib/python3/dist-packages`, which is also understood by `pip`. By default, `/usr/local` overrides `/usr` in the module loading mechanism. The creation of a virtual environment is possible with `python -m venv <directory>` or alternatively `virtualenv <directory>`. It is then supplied with its own copies of the Python interpreter and package installer, and using them confines all installations to within the specified directory, specifically into the relative subdirectory `lib/python3.`X`/site-packages`.

The command `pip list` shows all installed packages, and a subsequent `pip show <package>` shows details including about where they are installed.

[13] Virtualenv website: `https://virtualenv.pypa.io/en/latest/`
[14] PEP 668: `https://peps.python.org/pep-0668/`

New packages can in principle be found with `pip search <term>`, but over time this has caused too much load on the servers, and therefore lookups now need to be done interactively on the package repository websites.

Python packages may depend on other packages such that installing a single package triggers the implied installation of many other packages. These are either pure Python packages that are simply placed into the corresponding interpreter folder, or native packages that are compiled on the spot in order to work on the computer's hardware architecture. There is only one category of dependencies with such packages, so that excluding rather optional ones is not possible; however, it is possible to exclude all dependencies for tests with the flag `--no-deps`.

Packages may not only contain pure Python code but also natively compiled code in other programming languages such as C. In this case, the installation will automatically trigger the compilation and placement of the resulting shared libraries. As a prerequisite, a C compiler along with other build tools (assembler, linker) needs to be present.

The canonical invocation for user-wide package installation from the global Python Package Index (PyPI[15]) follows the format `pip install <package>`. Popular packages for data science include for instance `pandas` or `dsfaker`. For better control over the versions, a concrete version of a package may be specified (`pandas==2.0.0`), or a range of permissible versions may be given. In more complex scenarios requiring multiple packages, it is a convention to create a `requirements.txt` file with one versioned dependency per line. The installation of all packages can then be automated with one command: `pip install -r requirements.txt`.

Details on using pip to install packages are documented on the Python website.[16] The complementary view on how to produce such packages from own code are documented as well.[17]

Commands to repeat in alphabetic order: pip, virtualenv

5.7.2 Advanced Python package management with Pipx and Poetry

While `pip` handles the heavy duty tasks of package management, by itself it is often insufficient for the needs of complex yet productive data science environments. Two extensions are available to ease the installation of applications and the handling of dependencies.

[15] Python package index portal: `https://pypi.org/`
[16] Pip user documentation: `https://docs.python.org/3/installing/index.html`
[17] Pip package production: `https://docs.python.org/3/distributing/index.html`

With `pipx`[18], executable Python applications can be installed from various sources that run in non-privileged, isolated directories on a per-user basis. This essentially combines the functionality of virtual environments, package installations and taking care of custom `PATH` environment variable settings, so that executables from the packages can be invoked directly in the shell. Moreover, `pipx` supports continuous development scenarios by referencing code repositories. For instance, to install the Datadiary package that summarises the outcome of machine learning processes, the command can be invoked to proceed with the installation directly from a Git repository: `pipx install git+https://github.com/itsayellow/ datadiary`. Hence, `pipx` is applicable to all Python packages that provide executables on the command line.

From the user perspective, the installation is therefore as easy as it can get. Still, providing own packages takes some effort in describing the packages, formulating dependencies and performing the publishing process on package repositories. With Poetry[19], providing custom Python packages is made easier in contrast to the conventional approaches of distutils/setuptools (`setup.py`, `pyproject.toml`). It allows for fine-grained specification of dependencies, caching during dependency resolution and publishing to PyPI or other package distribution sites.

Additional projects and distribution channels for Python packages exist, including Conda and the C++ reimplementation Mamba. They can be consulted when needed.

Commands to repeat in alphabetic order: pipx, poetry

5.7.3 Package management for other programming languages

Beyond Python, many programming language communities have developed ecosystems to distribute software. In typical data science scenarios, it is common to mix applications and libraries from these ecosystems. Through files, databases, services and containers, different software implementations can interact with each other despite differences in the implementation language. The following thus summarises how to install packages written in popular languages, using the respective mechanisms for selection and deployment.

Projects using R can rely on the comprehensive R archive network (CRAN)[20] to find additional packages. They are organised into libraries, represented by local directories. The syntax for the installation of a locally available package (archive file `packagename.gz`) from the shell is `R CMD INSTALL -l`

[18] Pipx website: `https://pypa.github.io/pipx/`
[19] Poetry website: `https://python-poetry.org/docs/`
[20] CRAN: `https://cran.r-project.org/`

`/path/to/library <packagename.gz>`. There are also alternative techniques to download packages directly from within an R script or interactive session, and those can also be combined with the shell invocation to access CRAN with the command: `R -e "install.packages('<packagename>')"`. An exemplary invocation would be with the `LGDtoolkit` package. By default, R installs packages system-wide into the directory `/usr/local/lib/R/site-library` and therefore requires privileged invocation with `sudo`.

JavaScript packages based on the NodeJS runtime are distributed via the Node Package Manager (NPM[21]). For instance, installing the JavaScript equivalent of the Pandas module is conducted with the command `npm install pandas-js`. Apart from the code, such packages have a package file in JSON format with metadata and recursive information about their own versioned dependencies, in two sets: for running the code and for development. When such a file exists in a project directory, simply calling `npm install` automatically covers all listed dependencies, effectively preparing the project for use. In contrast to Python packages, NPM packages are installed in a per-project scope by default into a `node_modules` folder. The flag `-g` activates a global, system-wide installation, with the location `/usr/local/lib/node_modules`.

Java artefacts are packaged with Maven[22] and distributed via Maven Central[23]. The `mvn` tool build prepares the project and ensures the download of Maven dependencies. Alternative approaches exist, such as building Java projects with Gradle[24]. In general, these approaches are more software engineering-centric and are typically used within other Java development project structures, regulated via files such as `pom.xml` or `build.gradle`. This also applies to other compiled languages such as C and C++, where build commands like `make` or `cmake` are common, indicated by the presence of either a `Makefile` or a `CMakeList.txt`. If these build files do not yet exist, they may be produced by a locally existing executable auto-configuration script `configure`, which produces these files from templates based on the *Autotools* framework. If even that script does not exist, it might be possible to bootstrap it with another script, by convention called `autogen.sh`. C/C++ projects are more scattered over the Internet, although recently, with the C++ package repository[25], a package catalogue with dependency tracking support has become available.

Commands to repeat in alphabetic order: cmake, make, mvn, npm, R

[21] NPM website: `https://npm.io/`
[22] Maven website: `https://maven.apache.org/`
[23] Maven package search: `https://search.maven.org/`
[24] Gradle user guide: `https://docs.gradle.org/current/userguide/userguide.html`
[25] C++ repository: `https://cppget.org/`

5.7.4 System package management with APT

Not all applications and tools are implemented in Python or one of the other covered languages, and even some of those that are require a deeper integration into the operating system. Additionally, it is convenient to have a unified, language-independent way to install well-curated packages whose dependencies are balanced out.

Hence, an OS-specific software catalogue is needed. On Debian or Debian-based systems such as Ubuntu, the Advanced Package Tool (APT) performs this role. It does so in conjunction with a well-maintained, tightly integrated and policy-driven repository containing packages around the OS itself as well libraries, applications, datasets and documentation. Thousands of packages including their dependencies can be installed with a tool invocation of the form `sudo apt-get install <package>`. The command looks up package locations in a system-wide catalogue cache configured with a sources list file `/etc/apt/sources.list` containing references to installable packages (`deb` lines) and, if properly configured, corresponding source packages (`deb-src`). An example line for Debian is `deb http://ftp.ch.debian.org/debian/ bookworm main`, indicating the named version of the OS distribution and the main archive that contains exclusively free software.

Before the installation succeeds, the system-wide catalogue cache must first be created in the directory `/var/lib/apt/lists` via the command `sudo apt-get update`, which is also updated occasionally with the same command as the repository evolves. Rarely, for finalising a system and making it immutable, it can also be deleted again `rm -rf /var/lib/apt/lists`. The installation command itself also caches packages, which fill the disk over time, in the directory `/var/cache/apt/archives/`. They can occasionally be cleaned up with the more accessible command `sudo apt-get clean`. Packages are archive files created with `ar` and containing `tar` files. Apart from metadata and installation scripts, the main file `data.tar` contains directory hierarchies that are applied relative to the system's root directory `/`.

A typical example would be `curl` in order to ensure the system-wide availability of this command that is a Swiss army knife for interacting with web services. However, while `pip` works both in user mode and in privileged system-wide mode, `apt-get` installation and maintenance commands require system privileges, i.e. the use of `sudo`. There are some ways around this requirement. The first way is to retrieve only the installable binary package with `apt-get download <package>` or the corresponding source package with `apt-get source <package>`, if the proper source lines were added to the sources list. As a downside, this way skips the dependencies. The second way is to use a fake root environment, for instance, by working entirely within a user-level OS hierarchy through the command `fakeroot -s fakechroot.save fakechroot debootstrap --variant=fakechroot book-`

`worm /tmp/osroot`. This results in a the directory `/tmp/osroot` containing an entire operating system, which can then be activated with the follow-up command call `fakeroot -i fakechroot.save fakechroot chroot /tmp/osroot/ /bin/bash` to obtain a shell with fake root privileges. The third and most complex way, apart from the use of non-fake virtual environments with `chroot`, is virtualisation or containerisation to obtain virtual root access.

This requirement for privileged invocation does also not apply to simple package description search through `apt-cache search <term>`. A related command is `sudo apt-file update` that creates a system-wide database of mappings of file paths to packages containing those paths. While it requires root privileges, the subsequent search command `apt-file search <partialpath>` does not.

If the sources list gets modified to include a new version of the distribution, a more complex upgrade command in the form of `sudo apt-get dist-upgrade` should be run to accomodate larger changes in the packaging.

Despite the high curation quality, the package updates sometimes get stuck. Diverse calls such as `apt-get -f install` or to the underlying Debian package management tool (`dpkg`) may help in resolving the issues.

Over time, one learns the package-naming conventions, such as `python3-`X for Python packages often taken from PyPI or `r-cran-`X for all R packages taken from CRAN. The online package catalogues can also be searched, for instance, for Debian[26] or Ubuntu[27], which in contrast to local system searching is especially useful to find packages not currently offered for the running distribution version.

Commands to repeat in alphabetic order: apt-file, apt-get, debootstrap, dpkg, fakechroot, fakeroot

Repetition

1. In case the editor Vim is not yet installed on a system, how would it be installed?
2. With which command can a user count the number of externally installed Python packages?

5.8 Container management

Application isolation and containerisation has become widely popular since the mid 2010s due to a number of convenient technology stacks. Isolation

[26] Debian packages: `https://www.debian.org/distrib/packages`
[27] Ubuntu packages: `https://packages.ubuntu.com/`

itself has for many decades been confined to file systems with the change root (`chroot`) command, but has been expanded to other OS-managed resources with *CGroups*, which paved the way for convenient containers. Containers are therefore a fine-grained runtime isolation mechanism at the OS level with representation in `/sys/fs/cgroup/`. At the same time, tangible container images have complemented the isolation with greater portability and more options to ship code and data from creators to users.

The most widely used containerisation stack and distribution channel especially for mainstream use by data scientists is *Docker* and the Open Container Initiative (OCI).[28] OCI is an evolving set of specifications on container images, runtimes and distribution mechanisms, and Docker Hub the most popular public distribution site. In recent years, *Podman* has emerged as mostly-compatible container interaction tool with security benefits. It has simplified configuration requirements and integrates into the operating system process management without requiring elevated privileges, allowing for better self-healing capabilities not available in the plain Docker client.[29] This part of the book introduces primarily Podman, explains how to work with existing containers, and also informs about how to create custom containers for isolating and shipping own code and data. Occasionally, references are made to Docker and to OCI specs.[30]

5.8.1 Introduction to Podman

Podman consists of the intuitively `podman` command for individual container and container image management. Docker moreover provides support to operate composite applications through Docker Compose and Docker Swarm. Due to operational complexities with the Docker stack, there are alternative runtimes for the same container images, with some like Podman almost offering a drop-in interface, and others that operate slightly differently.

With Podman, the other package management approaches (programming language-specific and OS-level packages) are complemented by the ability to distribute complex applications in a pre-configured way encapsulated in container images. At the target machine, only some system bindings such as port numbers (for services) or volume directories (for applications requiring data persistence) need to be set up. Containerised applications can therefore run multiple services implemented in different languages without interfering with the host system such as a data scientist's workstation or virtual machine. A downside of this approach is the duplication of code in dependency libraries, which also requires discipline to keep container images updated to not be ex-

[28] Docker/OCI ecosystem: `https://opencontainers.org/`

[29] Podman documentation: `https://docs.podman.io/en/latest/`

[30] OCI runtime specification: `https://opencontainers.org/release-notices/v1-1-0-runtime-spec/`

posed to security issues. The `podman` command offers many subcommands, each with its own set of options and arguments. In the following, only the most-needed commands are explained.

5.8.2 Fetching and running containers

Container images use a layered file format to store containerised application code and data. The user workflow consists of fetching the layered images from a container registry, combining them into a local directory tree representing an entire filesystem hierarchy (minus OS kernel), and then executing commands within that filesystem through CGroups isolation. Docker Hub[31] is the default public registry on which thousands of container images are available. Collaborative programming platforms also offer integrated container image registries, and, moreover, it is possible to operate standalone instances for private use.

Finding the right image on any registry requires searching, trying out and trusting the provided images or the provider behind them, and gradually building up experience on what to look for. Image names on Docker Hub usually consist of one or two stem components (user name being the first one which could be omitted for official images) as well as a version number or practically unique version hashsum. With `podman search <registry>/<term>`, an approximated search for published images can be conducted, with the same caveats applying to Python packages and OS packages.

The canonical form of invoking Podman interactively is by specifying the name of the container image, optionally extended with the version information and the launch command, as in: `podman run -ti <container-image>[:<version>] [<command>] [<arguments>...]`. By default, the start command given at container image build time is used, but it can be overridden. In particular, debugging an image that does not start as intended is often possible by overriding the default start command with a shell invocation: `podman run -ti ... /bin/bash`. The run command implies a `podman pull <registry>/<container-image>` with all images stored as overlay file systems in the per-user path `~/.local/share/containers` in case there is no local image replica yet. If there is, pull can also update from a moving version tag.

If the image does not yet exist locally on the machine the tool is executed on, its layers are transparently downloaded and merged, leading to a slight delay in invocation. For testing purposes, a `hello-world` image is provided on Docker Hub that runs without further arguments and exits after delivering a message, and `alpine` is another convenient and small image to get a basic containerised shell. An example invocation of a useful container with configurable launch command would thus be as follows: `podman run -ti curlimages/curl:latest https://www.zhaw.ch/de/hochschule/`.

[31] Docker Hub: `https://hub.docker.com/`

Port numbers (`-p <hostport>:<containerport>`) and directories serving as volumes (`-v <hostdir>:<containerdir>`) connect the container's OS resources to the one from the host OS. For non-interactive invocation, the flags `-ti` are dropped, and the new flag `-d` leads to an invocation in the background, informing about the container identifier (id) as only output.

With `podman images` and `podman rmi [-f] <image-id>`, container images can be managed. Further commands such as `podman ps` or `podman kill <container-id>` are available to monitor containers and to manage the lifecycle of containers beyond starting up. Their semantics has been derived from the respective OS-level commands. In case a container has exited, `podman ps -a` shows an archived list of previous executions that still remain until `podman rm` is invoked on them. Well-maintained container images also provide endpoints for tracking the health status, showing up in the status column of the process list, in conjunction with specifying a container restart policy as self-healing instrument. With advanced container orchestrators, further self-management, autoscaling and other operations-supporting features become possible.

5.8.3 Building custom container images

A container image is built by incrementally modifying an existing image and storing the differences as another layer or set of layers. In the most trivial case, this would mean adding some files and customising the default startup command. Hence, selecting a base image to start with becomes a crucial task and a skill to master.

From a security viewpoint, often only a subset of 'official' images are considered trusted and are used exclusively as a basis for deriving custom container images. For example, instead of directly running the *Azure ML Inference* container, one would fetch an appropriate base image such as *Alpine Linux*, and then install the Machine Learning (ML) tools such as Tensorflow, PyTorch and Scikit-Learn manually into a derived image. Naming (tagging) the images for private use, uploading them to a private registry to share, and uploading them to Docker Hub for world-wide access are further options to consider after having modified an image that way.

Building such derived images works by specifying the base image as well as any modification commands in a `Containerfile`. This file can be written manually, or for some types of application even be generated automatically, at least to some degree. A Containerfile consists of a specification of the base image (`FROM` tag), files to be copied or commands to be run within the image (`COPY` and `RUN`, respectively), and the default launch command (`CMD`). The build command is then `podman build -t <tag> .`, with the last dot being a reference to the current directory which contains the Containerfile but also all files referenced from it.

Commands to repeat in alphabetic order: chroot, podman

Repetition

1. Can Podman be used to run a different version of the operating system?
2. How can PyPy be run interactively in a container?

5.9 Data management and version control

Keeping track of data, code and configuration files requires data-centric tooling for basic shuffling of data between locations. Before starting to accumulate data, certain criteria need to be assessed. This concerns primarily the protection of sensitive data. Such data contains privacy-related identification of persons or business secrets. It should never be stored on unencrypted physical media such as disks and should never be exposed to the Internet. In contrast, in order to not lose valuable data, safety replicas and backups need to be arranged. Automating this process may require lifting the exposure criterion for a short amount of time, still with the aim to minimise risks. This minimisation also includes occasional replacement of physical media and regular checks on the presence and recoverability of data. Once the criteria on data safety and security are fulfilled, the question becomes more technical concerning the best tools to use for content-agnostic data management on the file level.

Over the years, delta transfer and version control systems have emerged as technologies of choice for software engineers but also for data scientists with potentially large files. From early centralised approaches in version control such as CVS and Subversion (SVN), the collaborative nature of many projects has led to the proliferation of decentralised approaches. The widespread adoption and commercial success of Git has led to it being the de-facto standard for data management especially in development and operations, whereas unversioned data might also be effectively managed with RSync. This section explains both tools to ensure that file-based data can be handled in a safe manner.

5.9.1 Delta synchronisation with RSync

The previously introduced commands to copy files have basic support for incremental synchronisation and backup. Using `cp -auv <source> <target>`, or `--archive --update --verbose` in the long form, copies all files from a source directory including their metadata but only if they have changed compared to the last run, based on a comparison with the contents of the (previously existing) target directory, and informs about the progress of copying. Running the same command twice while keeping the source files unchanged shows that no

copying takes place anymore on the second run. Copying the files to a remote machine requires a syntax like `scp -r <source> <machine>:<target>` that unconditionally performs a copy operation without consideration of whether changes have occurred.

In summary, while `cp` and `scp`/`sftp` are suitable to copy few files and directories locally and remotely, respectively, the remote copy process always assumes the full file, leading to unnecessary occupation of bandwidth and processing time. When the amount of data is large and the amount of modified data is small, it is more practical to synchronise only the changes, represented as deltas between the old and the new version. With `rsync`, delta transfer happens transparently both on local systems and across the network. RSync is therefore a suitable single and efficient interface for all data shuffling, no matter whether system boundaries need to be crossed.

A typical invocation is `rsync -avz <source>[/] <destination>[/]`, with the flags representing archive mode (keeping all file attributes), verbosity, and compression of the deltas. The trailing slashes are significant concerning the reference to the directory itself or its contents. Source and destination either refer to local directories or to remote ones based on SSH in the form of `<user>@<machine>:<path>`. In case certain file and directory patterns should not be part of the replication, the option `--exclude <pattern>` (only available as long-style option) can be added to the command.

If low copy time is less of a concern compared to low bandwidth use, RSync supports a bandwidth limitation in the form of `--bwlimit <rate>`. The rate is typically specified with a data size per second unit suffix, for instance as `2.3m` to allow for 2.3 MB/s. Units can be chosen flexibly, including `g` for GB/s which is closer in line with evolving network speeds. For comparison, `scp -l <rate>` requires the value to be specified with high numbers in kB/s, whereas `cp` has no bandwidth limitation support, making RSync also a suitable choice overall for predictable rate data transfers independent of local or remote operations.

RSync can also run in daemon mode on a server to grant public read access to files organised as areas, offering a delta-capable alternative over the File Transfer Protocol (FTP) or HTTP. In that case, a listing of the exported areas can be obtained using `rsync <machine>::`, and the source location in the copy process is then addressed as `<machine>::<area>`. In contrast to version control systems, datasets obtained that way will be smaller albeit not collaboratively editable.

File transfers with RSync are documented on its website.[32] While RSync is not a backup solution in itself, several backup tools and strategies are built on top of it, such as RSnapshot[33].

[32] RSync website: `https://rsync.samba.org/`
[33] RSnapshot website: `https://github.com/rsnapshot/rsnapshot`

5.9.2 Version control with Git

Git is a tool to shuffle files between machines, but also a Version Control System (VCS) to track the evolution of those files, containing data and software with version history and a clear indication of what changed. Using Git properly aids in distributing and backing up data, ensuring provenance of data, code and models in larger projects and traceability of modifications.

The central data structure of Git is a *repository* containing binary files with the full history of all tracked files. Repositories can be cloned (replicated) and delta-updated multiple times from each other, leading to a decentralised model, which is, however, in practice often subject to agreeing on one centralised master repository, not for technical but for social or legal reasons. Repositories grow over time, but due to delta storage of the modifications, the growth especially with text files is modest. The advantage of keeping the entire history on all machines the repository is cloned to is that most operations such as search through the history work locally without requiring permanent network access. Especially for mobile workstations, this is a practical design choice. Each file tracked in a repository exists in versions that start with the addition of the file and run up to the lastest version. These versions might be located in different branches, so that each branch contains a subset of versions. The default branch is often called `master`. Version numbering in Git is not consecutive but based on file content hashsums, which are hexadecimal numbers often abbreviated as long as the short version is unique within a repository (e.g. `510906fa` as shortcut for `510906faecd6e8eae8c74b1cea9294347b3f1a97`). The latest version of each branch with all latest versions of the files in that branch is referred to as `HEAD`.

The second data structure of Git is the *index*, which memorises intended changes to the repository. An index is a local companion to a clone. The third data structure is the *working directory*, essentially the directory in which the files representing the currently checked out version of a branch in a repository are located and are being worked on. A working directory may therefore be either clean, or consist of a number of untracked files and/or files currently marked for changes (addition, removal, modification) in the index.

5.9.3 Basic usage of Git

The Git usage from a data scientist perspective follows a number of workflows centered around working on a cloned repository, adding files to branches within the index, commiting those files to the repository locally, and propagating the modifications to other repository clones while also ingesting modifications from them. Delta transfers work similar to RSync, with the advantage of versioning, but with the slight disadvantage of requiring double the space on checked-out working copies.

The standard command to invoke the Git client is just `git`, with a plethora of subcommands. There are also graphical frontends to Git available such as Github Desktop, and many collaborative environments automatically synchronise changes through Git. Often, Git is not yet properly preconfigured upon first use. The basic configuration requires two commands: `git config --global user.email <email>` to set a contact address, which may not necessarily be an e-mail address, and `git config --global user.name "<name>"` for a corresponding full name. The global setting ensures that the configuration becomes usable across all working copies. The configuration settings are then stored in the file `~/.gitconfig` which with some experience can also be edited directly to modify the identity, add custom commands and perform other settings.

When using Git on the shell, an apparent characteristic is that all files are tracked in branches of local repositories, effectively represented by hidden directories called `.git` at the top folder of a checked out clone, i.e. working directory, also called working copy or workspace. The repositories are initially empty when created with `git init` within an arbitrary directory that then becomes a working copy. Repositories can also be created with `git init --bare` in a dedicated directory meant purely for cloning without being usable as a working copy. When a repository already exists, it can be cloned with `git clone <url>`, with the URL either being a local file path, an HTTP(S) URL to a web-served Git repository or an SSH account on a server managing Git repository access based on SSH keys.

In a working copy, files can be prepared to be added to the default branch, typically called *master*, by marking them with `git add <file>`, which only adds them to the local index in an intermediary step. Directories are tracked implicitly as paths to those files, while empty directories are not tracked. Nevertheless, the `add` command also works recursively if a directory is specified. Only directories and files matching the names or patterns given in a `.gitignore` files are excluded. That file can be placed in the top-level directory of a repository or in subdirectories to override the configuration for those. Consequently, `git rm <file>` removes files or recursively removes directories again. To maintain an overview, `git status` shows information about the branch, the index and untracked files.

The current branch can also be queried with `git branch`, with the option `-a` to show all available branches. If desired, to try out experimental changes, a new branch can be created with `git branch <name>` and switched to with `git checkout <name>`. Whenever the working copy ends up in a broken state, for instance, due to versioning conflicts, it may also be disassociated from any branch, requiring more work to be aligned again. In those cases, `git merge` with the default merge strategy or a custom one helps to recombine independently conducted changes across branches or from different people.

A repository as well as the index both contain hashsum references to files. Whenever a file changes, the hashsum no longer matches, and the file is detected as modified. The modifications can then be staged to be added to the repository as well by invoking `git add` once again, keeping the index current. The actual modification of the repository based on the index then requires another command, `git commit`, which produces a new revision. In this command, a commit message can be entered either interactively or for short messages via `git commit -m "<message>"`. To combine the two steps (adding to index and finally adding to local repository), `git commit -a` can be used.

The history of all changes on the current branch can be shown with `git log`, and all changes themselves with `git log -p`. This command also shows whether all branches and remotes are synchronised. Remotes refer to the machines that are known to host other clones of the repository. When a working copy is initially cloned from an existing repository, that one gets recorded as known. Other clones can be registered manually with `git remote add <name> <url>` and queried with `git remote -v`.

Git then becomes decentralised by the ability to push revisions on a branch, in other words: to synchronise a branch, with remote copies of the same repository. Hence, a data server may run a Git daemon process or SSH-wrapped Git executable to receive modifications via the `git push <remote>` command, where the remote is a symbolic name usually pointing to an HTTP or SSH URL. Calling only `git push` attempts the default remote. If the modifications are purely additive, the push command is accepted and the repository branches are synchronised. Otherwise, for instance, after concurrent modifications by multiple users, the user intending to push is instructed to resolve the conflicts first. This usually requires merging the user's changes with those of other users by running a sequence of `git pull`, manual resolution via file editing, and finally again adding, committing and pushing.

A guided introduction to Git is given in the 'Pro Git' book. Further information including client download options are available on the website.[34]

5.9.4 Advanced usage of Git

Git offers the ability to execute repository-side hooks upon receiving modifications. This feature can be used to prevent the modification in the first place, by executing sanity checks on the files, as well as to trigger external actions such as sending the current files to an external service. Being activated by repository changes applies both to the local repository (after `commit`) and to any attached remotes (after `push`). This ability allows for setting up continuous delivery schemes. For instance, a data scientist may train a model, and pushing the files to Git ensures that the model is also properly deployed in

[34] Git clients: `https://git-scm.com/downloads/guis`

all inference services. On remotes, the `pre-receive` and `post-update` hooks are primarily of interest for this functionality to enforce policies, whereas in the working copy, the `commit-msg` hook can perform a first sanity check on what got modified, and the `pre-push` hook can further check whether a series of commits are worth pushing and likely allowed by the remote's policy. The hooks are implemented as shell scripts invoked by Git. All of those scripts must be placed as executable shell scripts into the repository's hook directory, which is in `.git/hooks` relative to the top level of any working directory and by default already contains templates for common hooks.

Git is optimised for text content. Smaller binary files can be stored without a problem. However, larger binary files such as compressed models or multimedia content not only become inefficient in terms of double space handling (for the checkout and the local repository) but also in terms of change detection and other Git management tasks. For this purpose, extensions are available such as Git LFS (Large File Storage), Git-Annex and DVC (Data Version Control). Git LFS stores only metadata about the files in the repository and can therefore inform about the absence of expected files. The files themselves are stored on a special LFS server. Git LFS must first be activated inside a working copy with `git lfs install`. Large files are then marked for tracking with the command `git lfs track <pattern>`, producing a `.gitattributes` file which must be added to the repository. Git-Annex follows a similar design, but does not require a dedicated server. Instead, it can synchronise large file content flexibly via RSync. The usage pattern is `git annex init` followed by `git annex add <largefile>` which replaces the file with a symbolic link to it, putting the actual file into the folder `.git/annex`, and marks the file within `git annex status`. The command `git annex sync` then synchronises repositories. Finally, `git annex copy --to <remote>` replicates the large files to a remote manually.

Commands to repeat in alphabetic order: git, git annex, git lfs, rsync

Repetition

1. Which merge strategies are available in Git to resolve conflicts after a pull?
2. You would like to dive deeper into extended file system attributes and have an idea to extend the custom `attr` tool for that, hosted at `https://git.savannah.nongnu.org/git/attr.git`. Which steps would be needed?

5.10 Data processing tools

Programming languages such as Python allow for very powerful text processing. Yet sometimes, as part of a shell-level automation workflow, it might be more convenient to remain on the shell for trivial processing steps, or it might take too much effort to implement a certain text processing algorithm.

Coreutils are therefore available as a collection of several small tools invoked from a shell to facilitate the processing of data stored in files. This encompasses the formatting of data, the creation of checksums and statistical information about file contents, performing string-based and numeric processing as well as search. A subset of the functionality is briefly summarised here, in conjunction with Sed, Grep and other tools also suitable for command-line data processing. While individual commands are documented with manual pages, a more systematic documentation is often provided in the so-called info format, especially for the individual utility programs that are part of Coreutils; hence, `info coreutils` brings up relevant pointers. A broader introduction to use coreutils and other command-line utilities for data science tasks is given in online books.[35]

5.10.1 Text search

The `file` command is able to inspect and inform about the content of any given file. If it is a text file, in contrast to a binary file, the text encoding is also determined heuristically. The default output is for human consumption, whereas a technical representation as MIME type is given with the parameter `-i`. A file may also be reported as a symbolic link; in that case, using the option `-L` to dereference the link is advisable. Working with text data files from various sources may also require a more sophisticated way to determine the encoding. Although some text editors are capable of showing a text file encoding, this is always subject to heuristics. The `chardet` tool includes a confidence value to make this decision more transparent.

Text files can be summarised with `wc` to count lines, words and bytes. One should be careful not to mistake the bytes with characters, as depending on the encoding the number might be different. Hence, to determine the number of characters for instance to check form limits of significance to humans, the option `-m` or long `--chars` for character counting should be used. Again, care must be applied to not use `-c`, which unintuitively is the short form of the default counting in `--bytes`.

The canonical tool to find occurrences of text in unstructured and semi-structured files is `grep`, along with similarly invoked tools such as `ack`. These tools can also search directory trees recursively and report the findings in detail

[35] CLI data processing tutorial website: `https://datascienceatthecommandline.com/2e/chapter-1-introduction.html`

or summarised as binary result whether or not certain text has been found. Searching a text token in a file works by calling `grep <token> <file>`. All complete lines containing the token are printed to standard output, and if the token was found at least once, the command exits with code 0, or else with 1. The commonly used option `-q` (`-quiet`) suppresses the printing. By default, search is case sensitive, whereas case-insensitive search can be instructed with `-i`, so that searching for either `word` or `WORD` matches either occurrence in a file.

To search through an entire directory recursively, the option `-r` is used. When the directory traversal encounters a binary file, it also attempts a search there and may report a match if by chance a short sequence of letters also appears therein. To prevent this behaviour and skip binary files, the option can be extended to `-rI`. With `ack`, recursion is the default when either no file or a directory is given as argument to search in.

The dot in a search term stands for arbitrary characters; hence, to search for a dot, it must be double-quoted once for the shell and once for Grep itself, by using either `grep "\."` or `grep \\.` as search command. A number of characters are interpreted as basic regular expression but in a Grep-specific interpretation. To achieve support for standard expressions, the switch `-E` for extended regular expressions (e.g. `[[:digit:]]`) or `-P` for Perl-compatible regular expressions (e.g. `\d`) should be used.

Text search is slow when conducted repeatedly. There are fulltext search engines such as Xapian that depend on prior indexing of content that is not supposed to change afterwards, such as archive files, so that subsequent searches are faster. In practice, non-indexed search is fast enough for most purposes.

The `expr` command evaluates expressions related to text strings. For instance, the command `expr index abcdef cz` determines the first occurrence of any character of the character set `cz` in the string `abcdef` as a 1-based position, i.e. 3, whereas 0 would signal that no character from the set was found in the string. Alternative invocations are `expr match` for matching regular expressions, `expr substr` to find out substring relationships, and `expr length` to calculate the length of a string.

> Commands to repeat in alphabetic order: ack, chardet, expr, file, grep

5.10.2 Text processing

The text-processing commands work on files or text passed on standard input as their input and write the results to standard output, with the option to redirect them into other files through shell redirection techniques or explicit output file specification (often `-o <outputfile>`).

While `cat` and `tac` were already introduced for outputting text content, the `nl` command works like `cat` but prepends line numbers.

The `sort` command sorts a text file line by line, either alphabetically or, with the `-g` option, numerically. Sorting can be further configured to be case-insensitive (`-f`), strictly numeric (`-n`) and in reverse order (`-r`). To speed up the sorting of large amounts of text, the built-in parallelisation (`--parallel=`N) to use N CPU cores is advisable.

The related command `shuf` performs a random permutation of all lines, without repetitions by default or with infinite repetitions by the parameter `-r`. To reduce the number of repetitions, this option is usually combined with `-n <lines>`.

The `uniq` command drops all duplicate consecutive lines and with the option `-c` also counts the occurrences of each line in a file. To enforce that identical lines are consecutive, a text must be sorted first; a pipeline of the form `sort <file> | uniq -c` is therefore commonly used.

The `cut` command is able to extract separated columns. For instance, `cut -d , -f 1 <file.csv>` outputs the first column of a CSV-formatted file. The logical opposite is `paste`, merging content from several files as columns into a single output line per line. Similarly, `join` merges lines based on a common join key, dropping all lines without occurrence of the key across all files.

Sometimes, using `cut` is difficult because there may be one or more spaces used as separator between words. The `awk` command is an alternative in such cases as it can also extract based on token positions independently from repeated separators. Running `awk '{print $2}'`, for example, always prints the second word on a line, and `awk 'NR>10 {print $3}'` always prints the third word while skipping over the first ten lines (with `NR` being the number of records skipped).

Rearranging text to fit into a specific width can be achieved with `fold -w 20 <file>`.

Search and replace functionality based on regular expressions can be achieved with Sed. A typical use case for automated text replacement would be to substitute all occurrences of A with B in a file with the following invocation: `sed -i -e 's/A/B/' <file>`.

> Commands to repeat in alphabetic order: awk, cut, fold, join, nl, paste, sed, shuf, sort, unique

5.10.3 Numeric processing

Basic mathematical operations on integers are built into shells, not as commands, but as arithmetic expressions that can be evaluated. For instance, the expression $((1+1)) evaluates to 2, as can be verified with the command: `echo $((1+1))`. If `a` has the value 5, `$(($a*2))` returns 10. Supported operators include addition, subtraction, multiplication, division, exponentiation,

remainder (`+ - * / ** %`) as well as bitwise and comparison operators. Incorrect results can be expected with the division (`/`) which resembles the integer division operator (`//`) in Python.

For arbitrary-precision calculations, the `bc` tool can be used, taking an input formula on standard input. The standard math library needs to be activated to achieve the full functionality including floating-point numbers support. For instance, `echo 7/2 | bc -l` outputs the mathematically correct result, like all floating-point operations only subject to minor discretisation errors, and `echo "s(1.0)" | bc -l` returns the sine value for $X = 1$.

Moreover, several coreutils assist in numeric calculations. With `factor`, a number can be divided into its prime factors. For example, `factor 999999993` yields the factors 3, 17 and 19607843, all of which are prime. The `seq` command produces a sequence of numbers over which an iteration can be performed, similar to the `for` loop in Python.

> Commands to repeat in alphabetic order: bc, factor, seq

5.10.4 Media formats

Beyond text and numbers, files may contain multimedia information, often in binary format, such as documents, images, audio and video. Working with that kind of data requires format-specific tools. Typical operations include retrieving format-specific metadata, scaling, other manipulation and cross-format conversion. Due to the variety of formats, only a brief glimpse at the possibilities to automate those tasks in the shell is given here.

PDF documents are inspected with `pdfinfo`. They can be scaled with the GhostScript (`gs`) tool, as follows: `gs -sDEVICE=pdfwrite -dCompatibilityLevel=1.4 -dPDFSETTINGS=/<level> -dNOPAUSE -dQUIET -dBATCH -sOutputFile=<out.pdf> <in.pdf>`. The target level is one of `screen`, `ebook`, `printer` or `prepress`, in ascending order of quality. With `pdftk`, pages can be extracted (`pdftk <in.pdf> cat 2 output <out.pdf>`) or stitched together (`pdftk <in1.pdf> <in2.pdf> cat output <out.pdf>`). PDFs can be converted to text with `pdftotext`.

Raster images are usually stored as lossless PNG or lossy JPEG files. The `file` command shows resolution and colourspace. With `convert`, formats may be changed as well as the resolution and other settings, for instance: `convert -scale 800 <in.png> <out.png>`. Videos may be scaled with `ffmpeg`, which, despite the name, covers a lot of video formats through plugins, including MPEG/MP4, AVI, MOV and OGV. A typical invocation for resizing and quality adjustment in terms of data points per second for the video stream is `ffmpeg -i <in.mp4> -s 800x600 -b:v 1000k <out.mp4>`. Frames can be extracted from videos as still images with a command following this pattern: `ffmpeg -i`

`<in.mp4> -r 1 -s 800x600 -ss 10 -t 2 out-%03d.jpg`. This creates two images at seconds 10 and 11 of the video, and give each of them a consecutively numbered filename.

Commands to repeat in alphabetic order: convert, ffmpeg, gs, pdftk, pdftotext

Repetition

1. How can you look up all phone numbers in a collection of text files?
2. How can a picture be reduced in size to a quarter, i.e. with half of its original width and height?

5.11 Structured data processing

Several command-line tools exist to perform basic checks, queries and modifications on structured data files in the common formats CSV, XML and JSON. Although the choice of tools is large, their names are usually related to the data formats they can work on, making it still possible to find the right tool given a dataset and an associated task. Moreover, some commands exist to perform basic data science and machine learning on specifically structured files with tabular formats.

5.11.1 Format-specific processing

For all structured data files containing text content, the same advice applies as previously given for text-file processing. In particular, using `file` or `chardet` to determine the character set and encoding is recommended before starting the processing.

For JSON, `json_pp` performs pretty-printing (and basic validation). It reads from standard input and is therefore invoked either in the form of `cat <file.json> | json_pp` or, more elegantly, `json_pp < <file.json>`. The output is indented and dictionary keys are sorted alphabetically, giving a uniform representation that does not matter for programmatic processing but is more suitable for humans and for maintaining a JSON file in version control systems.

To learn more about JSON files and especially the evolution between multiple versions, the Pip-installable `genson <file.json>` generates a basic schema from a data file, which is again in JSON format following the JSON Schema specification.[36] The schema is written to standard output. A schema contains

[36] JSON Schema: `http://json-schema.org/`

a description of the structure and rules about the permitted values. Auto-generated schemas may be too strict or too relaxed and might therefore need manual refinement afterwards. With the tool `jsonschema`, such a schema (possibly refined) can be checked against various instances of JSON files (`-i`) whether or not they conform. Hence, the following sequence of commands should work by first generating the schema from one file and then checking that same file against the schema: `genson <file.json> > schema.json; jsonschema -i <file.json> schema.json`.

If there are multiple versions of a JSON data file, `jsondiff` creates a machine-readable (JSON-formatted) difference representation (diff) between pairs of two data files. The inverse command is `jsonpatch` that takes such a diff and the first file and reproduces the second file. Therefore, the following sequence should again work: `jsondiff <file1.json> <file2.json> > diff.json; jsonpatch <file1.json> diff.json > <file2b>.json; diff -u <file2.json> <file2b.json>`. The two files `<file2.json>` and `<file2b.json>` should then represent the same content, and if they are properly sorted, the text diff at the end is empty.

The `jq` tool queries over the nested list and tree structures that may exist in a JSON file. For instance, `jq .<key>` returns the value of a key top-level dictionary from a JSON file. More complex queries are possible by pipelines that resemble shell pipes. For example, to select dictionaries with key Y that are in a list that is defined by a top-level key X, use the following command: `jq '.X | .[] | select(.name=="Y")'`. Typical string processing methods can be chained as additional operators, for instance, `.name | contains("substring")`.

Linting and pretty-printing for XML files is done with `xmllint` based on XPath expressions. The equivalent of the example above with the modification that the key could be anywhere in the document structure would then be `xmlstarlet sel -t -v "//key"`.

CSVKit contains many small tools to work with CSV files. Using `csvclean` helps in preprocessing and removing common formatting glitches. Lines with a column matching an expression can be extracted with an adjusted grep command, for instance, on entries with price equal to 3: `csvgrep -d  -c price -m 3 <*.csv>`. The `csvtool` command is another way to work with CSV files.

Conversions between formats are also possible. With `csvjson <*.csv>`, a CSV file can be converted to JSON and with `csvsql` correspondingly to insertion statements for several relational databases. With `csv2`/`2xml` and `xml2`/`2csv`, bidirectional conversions between flat XML formats and CSV can be achieved. For instance, to convert a flat XML file to JSON, the following pipeline may be set up, assuming the source XML has an element $L2$ under its root representing the record which in turn contains $L3$ containing the field:

`xml2 < <file.xml> | 2csv L2 L3 | csvjson`. More complex conversions and transformations are the domain of data integration tools such as Meltano that will be presented in a later chapter.

> Commands to repeat in alphabetic order: csv2/2csv, csvclean, csvgrep, csvjson, csvsql, csvtool, genson, jq, json_pp, jsondiff, jsonpatch, jsonschema, xml2/2xml, xmllint, xmlstarlet

5.11.2 Training and inference

Tabular data is often the input for data analytics and machine-learning tasks such as training, testing and validation. Many of these tasks are conducted programmatically due to the sheer number of options and algorithmic decisions. Nevertheless, a few tools exist to perform basic tasks on the command line. One such tool is Vowpal Wabbit, which assumes training data in a line-based tabular format `<label> | <feature:value> <feature> ...` and test data in the same format with purposefully removed labels (that should be predicted). With Vowpal Wabbit, the training data is first used to produce an optimised binary mathematical model with `vw -d train.txt -f model.vw`. Next, the missing test labels are predicted in order with `vw -d test.txt -i model.vw -p predictions.txt`. The predictions can then be compared with the labels that were removed to determine an accuracy score.

Binary classification of text based on trained word frequencies (Bayes model) is possible with several tools, for instance, `spamoracle` for e-mails. First, training data is specified to contain either good or bad e-mails, as follows: `spamoracle add -good <mbox>` (and `-bad`, respectively). Then, newly incoming messages can be tested with `spamoracle test <mbox>`, giving a record with score between 0 and 1.

> Commands to repeat in alphabetic order: spamoracle, vw

Repetition

> 1. A RESTful API provides a compact JSON representation on a GET endpoint that should be formatted adequately for human inspection. Which command pipeline could be used?
> 2. A CSV file using semicolons as field diver has a column A with product names. How can all lines with A containing the character sequence *PRODUCT* be extracted?

Chapter 6

Middleware

Middleware, such as generic compute services, remote file systems, databases, message queues and message brokers as well as event processing frameworks, fulfil an important role of bringing scalable and stateful data processing capabilities to applications while keeping those applications lean. As opposed to rather static file management systems such as Git that only interact occasionally with servers, most middleware is operated continuously as standalone services, although some provide strong local processing support in the form of a tool as well or allow for configuring the service to listen only to local clients. From a programming perspective, for instance, from a Python application, it is therefore possible to connect to these services either locally or remotely to gain powerful functionality. This section covers middleware in a number of broad and partially overlapping categories: custom programmatic data and service provisioning, file system abstractions, database management systems, frameworks for data processing and integration, model serving and workflow execution.

6.1 Programmatic data serving

In addition to accessing web services from a client perspective, there is often the need to serve data and program logic from a service perspective. Serving means either to the human (from plain text to interactive websites based on structured and unstructured formats, i.e. HTML/XHTML) or to client applications (APIs or programmable web, i.e. structured data formats such as JSON, CSV, XML or YAML). Apart from the data formats and layout in the form of directories and subdirectories, the protocol needs to be defined. Data serving on the web is appropriate in most situations, giving the choice of both anonymous and authenticated access for read and/or write operations. Further protocols such

as FTP/SFTP, RSync or S3, or even decentralised protocols such as Bittorrent, each come with their own characteristics and can be chosen depending on the needs.

For static data serving on the web, simple tools like `netwox 125 -P 8080` or `thttpd` or even the Python built-in module `python -m http.server` can be used on the simplistic end of the spectrum and sophisticated web servers such as Apache HTTPd[1] on the other end. As explained in the section on system administration tools, long-running servers would be set up with service supervision and logging to ensure availability and to provide means to detect and correct any issues with the service provisioning. In static data serving, the endpoints for clients typically correspond to the physical file system layout, although advanced web servers offer support for virtual folders within virtual host definitions and interfaces to custom logic through Custom Gateway Interface (CGI) scripts.

Often, more complex logic elements such as differential data transfer and sophisticated endpoints are required. This leads to programmatic web-serving in which a user-defined application runs as service and waits for client requests. In Python, this functionality can be realised through appropriate modules. A few of these modules are briefly introduced in the next paragraphs. Which one to choose depends on the functional and non-functional needs as well as on the serving model.

6.1.1 Third-party module 'flask'

Several web frameworks make it easy to serve data, both of the static and of the dynamically generated kind. Data, including model files, can be made available over the network to clients on other machines, aiding the integration of pieces of software towards more complex software. One of the most widespread web framework is Flask[2]. In Flask, Python functions or methods are marked via decorators as accessible via the web protocol HTTP and interfaced on the server side with either an external web server or, for convenience in case scalability is not an issue, with the built-in Flask web server.

First, the flask object is created by calling `import flask` followed by `app = flask.Flask("Myapp")`. Functions serving as exported endpoints are decorated with the endpoint path and optionally with the list of supported HTTP method beyond the default of GET. Segments of the path may be typed to indicate that a field should be numeric or a wildcard path follows. For example, `@app.route("/hello")` or `@app.route("/<path:path>", methods=["GET", "POST"])`. The module import and use of decorators also works conditionally inside functions or methods in a class, using inner functions to make sure the

[1] Apache HTTP server: `https://httpd.apache.org/`
[2] Flask website: `https://www.fullstackpython.com/flask.html`

decorators only apply in case Flask is available. Finally, the application main loop is entered with `app.run(host="0.0.0.0", port=8080)` or a variation thereof, indicating the listening network interface determined by the IP address, and the respective port number. In this case, `0.0.0.0` indicates any interface, whereas `127.0.0.1` would listen only on localhost and thus prevent access to the service from outside the machine.

Within the methods, metadata and data can be accessed. For instance, the context-dependent attribute `flask.request.method` informs about the HTTP method used, and the method `flask.request.get_data()` gives access to any raw submitted payload in POST requests. Special methods also exist to process form data from HTML uploads.

For practical use, extensions such as Flask CORS and HTTPS activation need to be used to build applications that can be deployed and invoked across machines in secured environments. The CORS extension is imported with `import flask_cors` and instantiated atop a Flask application object with `flask_cors.CORS(app)`. Its behaviour can be further configured, for instance, through `app.config["CORS_HEADERS"] = "Content-Type"` to allow POST requests on JSON and other specific data types. HTTP is activated by passing an additional `ssl_context` parameter to `app.run()`. It refers to either a certificate/private key pair or an anonymous context. The first can be established through the Python SSL module with `context = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)` followed by the configuration step `context.load_cert_chain("cert.pem", "key.pem")` based on the two previously generated files, whereas the second is more trivial with `context = "adhoc"`. Both options nevertheless produce various security warnings in most web browsers unless a cross-signed certificate is used. More Flask extensions such as Flask Talisman[3] attempt to provide more security options.

Flask is primarily concerned with low-level HTTP request-response mechanics and less with the content of the messages. While HTTP responses can deliver HTML pages, their parameterisation calls for using Flask extensions that permit easy template rendering, such as Jinja[4]. Another extension is Flask-RESTful, easing the development of resource-oriented APIs.[5]

The command-line tool `flask` can help debugging a Flask application, when invoked with the following command: `FLASK_APP=x.py flask routes`.

Multiple alternatives exist with a basic functionality similar to Flask and similar integration into the web server gateway interface (WSGI) of Python, such as Bottle[6] or, specifically for continuous web sockets updates, Tornado[7].

[3]Talisman: `https://pypi.org/project/flask-talisman/`
[4]Jinja website: `https://jinja.palletsprojects.com/`
[5]Flask-RESTful website: `https://flask-restful.readthedocs.io/en/latest/`
[6]Bottle: `https://bottlepy.org/docs/dev/`
[7]Tornado: `https://www.tornadoweb.org/en/stable/`

6.1.2 Third-party module 'streamlit'

When not only raw data should be provisioned, but instead human-consumable visualisations, a data visualisation library can be helpful. Streamlit[8] is more oriented towards frontend prototyping and development. It is not often packaged yet and needs to be installed separately from PyPI through `pip install streamlit` in a virtual environment previously created with `python -m venv <venv-dir>`. The `streamlit` command documents its usage and gives examples, whereas the equally named `streamlit` Python module allows for integration into code. The idea behind Streamlit is that only the program logic is programmed and any visualisations emerge from that code, without data scientists having to worry about frontend/backend separation or content delivery via HTML.

For example, a Pandas dataframe created in a Python script can be visualised as a linechart with a single command `streamlit.line_chart(df)`, which renders an HTML page with embedded interactive chart. This script then needs to be executed with `streamlit run <script.py>` to let the visualisations take effect. The command launches a local web server on port 8501 and automatically opens a web browser window to it. Headless mode can be configured automatically by absence of the `DISPLAY` environment variable or by passing environment variables `STREAMLIT_SERVER_HEADLESS` or the option `run --server-headless true`.

More chart types are possible including `streamlit.bar_chart` and the generic `streamlit.pyplot` for anything manually plotted with Matplotlib. Tabular visualisation can be achieved with `streamlit.table` for static tables and `streamlit.dataframe` for dynamic tables. There are many integrations to visualise complex data structures such as maps, using the built-in `streamlit.map` function that requires geocoordinate columns in a dataframe (`lat`, `lon`) and defaults to using Mapbox or alternatively the more advanced `streamlit_folium` module, which directly interfaces with OpenStreetMap.

6.1.3 Third-party module 'bokeh'

Bokeh is another library aimed at human consumption of data. It integrates with Python data science libraries and generates visualisations as static files or as dynamically served web pages consisting of HTML markup and JavaScript code. Similar to Streamlit, it is not widely packaged and needs to be installed manually into a virtual environment through `pip install bokeh`. The submodule `plotting` then allows for creating the first visualisations. Similar to Matplotlib, a `bokeh.plotting.figure` object is created first (`p = figure(title="...")`), followed by plot commands such as `p.line(x, y, ...)` and finally `p.show()`. When invoking `bokeh <script.py>`, a temporary

[8] Streamlit website: `https://streamlit.io/`

HTML file is produced and the web browser is opened to display it. Bokeh also supports plotting circles and colourful scatter plots, colour maps, bar charts and stacked areas.

Bokeh can be combined with Flask by calling `flask.render_template` along with the utility function `bokeh.embed.components()` that returns both an HTML `div` element and a portion of JavaScript code (`script`, based on BokehJS) to embed into a rendered page template. This can be accomplished as follows: `flask.render_template("template.html", plot_script=script, plot_div=div, ...)`. Bokeh also ships with a standalone Tornado-based server running by default on port 5006. The command `bokeh serve --show <app.py>` deploys the application to the server and shows the generated web page in the browser on `http://localhost:5006/myapp`. Within the Python code, HTTP requests can then be accessed through the function `curdoc()`.

The reference documentation for Bokeh is available online.[9]

Repetition

1. Why does CORS support have to be added to web services that should be accessible from a dynamic web frontend?
2. Why can Streamlit-using scripts not simply be executed with the Python interpreter but instead require the `streamlit` command?

6.2 File system abstractions and network storage

Virtual file systems are the most basic form of middleware for hierarchical data storage. They provide a regular file system structure to applications while physically storing the files in arbitrary locations such as other file systems, network services or data encoding applications. Hence, the application does not need to take care of the specific interactions with any such service itself, including protocol and data representation details. Instead, the interface to the application is always the standard file system structure with directories, files and – with varying degrees of support – metadata. File systems in userspace (FUSE) are the basis for most virtual file systems on Linux. Some are well-maintained and can be used in production, while others are more experimental in nature. FUSE mounts run as processes so that the data availability depends on the availability of the service, which can be ensured with process supervision, e.g. SystemD units. Some FUSE implementations do not

[9] Bokeh website: `https://docs.bokeh.org/en/latest/`

handle all low-level operations on the file systems, but most applications write and read data from such mounts without problems.

6.2.1 Basic FUSE operations

The generic form for mounting FUSE file systems is via the `mount` command by specifying the type, like `mount -t fuse.<fusefs> [options] <source> <target>`, where the target represents a local directory and the source is a type-specific identifier such as a path, a URL or something else. File systems based on FUSE that are pre-configured by the system administrator in the file system table `/etc/fstab` can be mounted to the configured location within the local file system by privileged users with the command `mount <target>`. Usually, and especially for unprivileged users, the mount command name along with the type of file system (option `-t`) are also substituted by an executable with type-specific command name such as `<fusefs> <source> <target>`. To unmount again, the command `fusermount -u <target>` is used on Linux or `umount <target>` on Mac OS X. Such file systems can be layered in modular combinations, for instance, to achieve transparent compression, encryption and distribution of files. The application, which is unaware of these layered data modifications, writes into a file on a path of a mountpoint of file system A that encrypts it, and the underlying file system B delivers the data to an online service, for example.

Slightly confusingly, not all FUSE mounts show up in the `df` command output, but all show up when running `mount`. Unprivileged user mounts are by default only visible to those users, but the line `user_allow_other` in the otherwise almost empty configuration file `/etc/fuse.conf` along with the mount option `-o allow_other` changes this behaviour. This is also necessary for loop-mounted and bind-mounted file systems passed as volume to Docker containers due to the Docker daemon running under its own user identity.

Among the more commonly used abstractions are SSHFS, providing transparent file access over SSH connections, EncFS, adding encryption on the storage path, Fuse2FS, for loop-mounting disk images containing an ext2/3/4 file system, and BindFS, for bind-mounting one directory on top of another one, which may or may not be a FUSE mount location. Less commonly used but still useful are HTTPFS, allowing to interact with files on web servers as if they were local (unfortunately not supporting HTTPS), and Restic, for backing up data to various network services. An example for HTTPFS is to create an empty directory with `mkdir ~/cifs` and then mounting a website's HTML file to it with `httpfs2 http://cifs.servicelaboratory.ch/ ~/cifs`.

There are also more obscure abstractions such as π-FS or FUSE filesystems exporting the online mailbox as a virtual directory. Further information is

available online about the FUSE base technology[10] and, albeit incomplete, about selected file systems created with FUSE.[11]

6.2.2 Selected file systems and synchronisation

In case a user has access to a remote machine via SSH, then using SSHFS is a straightforward way to view a directory on that machine's file system. All files remain physically on that machine but can be listed and modified within the mountpoint. A production-grade invocation of SSHFS to resiliently mount the remote machine's `work` directory to the same name on the local machine is `sshfs -o reconnect,ServerAliveInterval=15,ServerAliveCo untMax=10 <server>:work ~/work` (on Mac OS X, where it supports slightly fewer parameters: `sshfs -o reconnect <server>:work ~/work`). This command mounts the remote folder of a machine serving SSH to the local equivalent and instructs SSHFS to overcome temporary connection drops, providing an always-on experience for remote file access. To automate this access without the need for manual authentication, apart from pre-confirming the host fingerprint with a first interactive SSH connection, a passwordless SSH public key needs to be deployed on the target machine, and an autostart file (e.g. via SystemD) needs to be provided. After mounting, the command `ls ~/work` works transparently on both the server (running the server's `ls` command on the local file system) and the client (running the client's `ls` command on the local mountpoint translating the access to the server's file system).

Disk images contain a file system within a file of fixed size and are a suitable mechanism to mount capacity-limited storage. They can be trivially produced by running `dd` followed by file system creation, for instance, with `mkfs.ext4`. Fuse2FS can then loop-mount these images: `fuse2fs <fs.img> <target>`. BindFS is able to mirror both loop-mounted and other directories to another location: `bindfs [-o allow_other] <source> <target>`. This makes the content accessible in two locations and can also be used to work around problems with low-level file system operations on the source mount.

EncFS and its alternative implementation, GoCryptFS, allow producing encrypted data from any application. When running `encfs <source> <target>`, a configuration mode and then a password need to be specified interactively. By default, any content is encrypted symmetrically with AES-256. The command `echo "plaintext" > <target>/text` leads to a 10-bytes unencrypted virtual file on the mounted target and a 26-bytes encrypted physical file in the source directory.

RClone builds on FUSE to provide a more integrated package for compression, encryption, chunking, as well as serving files both as local file system

[10] FUSE: `https://github.com/libfuse/libfuse/`
[11] List of file systems: `https://github.com/koding/awesome-fuse-fs`

and as locally operated file network service, while physically storing the files in arbitrary locations.[12]

Repetition

1. How can a filesystem contained in a single file be mounted?
2. SSHFS only grants access to certain files on a remote computer, not to the computer itself. Correct?

6.3 Database interaction and management

Databases (DBs) are useful collections of structured or unstructured data. The collection format can be very strict with schema information as in relational databases or rather loose as in schemaless/schema-flexible document databases. Database management systems (DBMS) assist in the collection process by providing data management commands and queries. Consequently, there are relational/tabular, tree-/graph-based and document-oriented DBMS as well as key-value stores, matching the respective database contents.

The term relational refers to normalised tabular data in which values in certain columns of a table form a reference to the same values in a column of another table. For instance, a table with addresses might contain country codes, and a second table maps these codes to country names. Tables are managed following the CRUD paradigm for creating (inserting), reading (selecting), updating and deleting rows. Read and update operations are also possible for a subset of columns. Read, update and delete operations are moreover possible on a subset of rows by filtering. Tables may be further grouped into schemas, databases and other higher-level structures. In non-relational databases, the main structures are collections or graphs.

Most DBMS offer a command-line client to manage tables, collections or graphs interactively and to insert or query data in the form of records or documents. For automated mass processing, programming interfaces specific to the chosen language are available, typically called connectors. Both clients and connectors require access to a running database service, although some also work on in-process embedded data structures.

6.3.1 Embedded relational databases with SQlite

The standard API from Python to relational databases is DB-API, although there is also support for such databases in Pandas due to the tabular data dominating in relational DBMS. Many DBMS require a complex server setup.

[12] RClone homepage: `https://rclone.org/`

For more modest use cases, an embedded DB entirely contained within one file is a good alternative. One such system is SQlite, an embedded relational DBMS with support for the standard query language SQL and the ability to interface from both Pandas and DB-API. All commands starting with a dot are non-SQL internal commands.

The canonical invocation on the shell level is `sqlite3`. Within its command prompt, a database file is created with `.open <filename>`. Then, tables can be defined with schema and populated, for example, with: `CREATE TABLE left (x int, y str); INSERT INTO left (x, y) VALUES (3, 4);`. The second value ought to be written as `'4'`, but the parser is flexible concerning the types. With `.mode table`, the default output formatting optimised for scripting is made more human-friendly, and a `SELECT * FROM left;` confirms the table contents.

From a Python script, that data can be imported into a Pandas dataframe by running a schema-dependent query. This is accomplised by importing the respective modules `import sqlite3; import pandas` and by running two successive commands `conn = sqlite3.connect(<filename>); response = pandas.read_sql("SELECT * FROM left", conn)`. The corresponding DB-API connection starts by setting up the connection first with `conn = sqlite3.Connection(<filename>)`, then creating a queriable cursor object on top with the query `cur = conn.execute("SELECT * FROM left")`, and eventually obtaining the query results through `cur.fetchall()`. This returns a list of tuples covering all columns, which should obviously only be used for smaller tables. Alternatively, one can iterate using `cur.fetchone()`, which returns one tuple at a time, or `None` at the end of the table.

SQlite also supports transient in-memory databases with a pseudo `:memory:` filename. The complete SQL syntax of SQlite is documented online[13], whereas the built-in Python module has its own documentation.[14]

6.3.2 Networked relational database systems

MySQL/MariaDB and PostgreSQL are relational DBMS that have seen active development and high adoption for many years. Both run either as standalone services or in various scalable cluster combinations. They support conventional relations but also binary-optimised structured data, as evidenced by the JSONB data type in PostgreSQL. MariaDB listens on port 3306, whereas it is 5432 (or merely a local socket in localhost connections) for PostgreSQL. The following describes an exemplary setup for PostgreSQL. First, the DBMS needs to be installed in the right version, for instance, with `sudo apt-get install postgresql-15`. Next, a user for local DB access needs to be created:

[13] SQlite syntax: `https://sqlite.org/lang.html`
[14] SQlite Python module: `https://docs.python.org/3/library/sqlite3.html`

`sudo -u postgres createuser $USER` and a corresponding database `sudo -u postgres createdb -owner=$USER <dbname>`. Finally, this database can be accessed interactively from the shell: `psql`. Automation is possible via the batch command execution `psql -c "<sql-statement>"`. In Python, DB-API is implemented by the widely packaged `psycopg2` module. A local database can be connected to with `conn = psycopg2.connect(database="<dbname>", user=os.getenv("USER"))`.

6.3.3 Beyond relational databases

The system families of key-value stores, document databases, graph databases and timeseries databases all offer means to manage data beyond tabular formats and relations. Many of the respective database management systems are however not easily installable or maintainable in operation.

Key-value stores offer low-latency access to dictionary structures held on disk or in memory. One of the earlier embedded database options for key-value storage was Berkeley DB. Later, networked alternatives such as Memcached[15] and Redis became popular. To improve network latency, they support setting and getting the values for multiple keys at the same time.

Document databases support collections of large structured and unstructured documents. In structured documents, such as JSON-serialised data, searches can be performed. Popular implementations beyond the mentioned document support in relational databases include MongoDB, CouchDB and BaseX (specifically for XML documents).

Graph databases represent graphs as labelled, weighted and directed relations between nodes. They implement graph algorithms such as shortest path search. Neo4J and OrientDB are typical examples of graph databases.

While conventional databases are suitable for mostly static data that needs to be retained over long periods of time, the nature of data is sometimes closely bound to its production time. Specific timeseries DBMS exist to handle such chronological data. Examples include TimescaleDB (an extension to PostgreSQL), as well as Prometheus and Victoria Metrics, both of which offer a native HTTP API for data management. One advantage of using a timeseries database is in being able to set up automatic compressions of events that have been longer back in the past, saving storage capacity.

TimescaleDB is installed into a running PostgreSQL extension by preparing the environment (`sudo apt-get install cmake libkrb5-dev postgresql-server-dev-15`) and executing the bootstrap script from the most recent version.[16] Next, the extension library needs to be loaded by setting `shared_preload_libraries = 'timescaledb'` in the global configuration file `/etc/postgre`

[15] Memcached: `https://www.memcached.org/`
[16] TimescaleDB downloads: `https://github.com/timescale/timescaledb/releases`

`sql/14/main/postgresql.conf` and activated within the client with `CREATE EXTENSION timescaledb;`. Afterwards, so-called hypertables on top of existing tables with a time column of type `TIMESTAMPTZ` are created with, for instance: `SELECT create_hypertable('<tablename>', 'time');`.

Repetition

1. Inserting a million records into a relational DBMS might be very slow. What could be done to speed it up?
2. Why does PostgreSQL not ask for a password when connecting to a local database?

6.4 Message brokers for real-time data processing

When data should be forwarded or processed immediately as soon as it arrives, message brokers are another form of middleware that is suitable for ephemeral and event-based scenarios. Events are received and need to be processed or distributed to other machines without necessarily storing them beyond the processing stage. This is typically called Event-Driven Architecture (EDA), Event Stream Processing (ESP) or, especially when certain state data such as counters and aggregate values are retained, Complex Event Processing (CEP). The encapsulated processing logic in this context is referred to as operators, which often contain standing queries applied to the incoming data-stream.

Message brokers are related to message queues and publish-subscribe systems. In all of these systems, there is a notion of events, event producers and event consumers. A variety of networked systems with overlapping functionality exist, for example, ZeroMQ, RabbitMQ, NATS, Kafka and Pulsar. Pulsar has the unique functionality that it executes small code functions to modify messages and influence the routing. ZeroMQ works daemon-less but also supports the setup of persistent devices to handle multiple dynamic producers and consumers. There are also file-based systems such as SEC, reacting on lines, for instance, continuously appended to log files, and Fever, listening to JSON events on a local socket.

In Python, ZeroMQ can be used with `import zmq`, setting up an event producer queue with `sock = zmq.Context().socket(zmq.REQ)` and connecting it to a consumer with `sock.connect("tcp://localhost:5555")`. The blocking consumer sets up its counterpart with `sock = zmq.Context().socket(zmq.REP)` and listens for producers with `sock.bind("tcp://*:5555")`. On TCP, acknowledgements for sent events need to be received; thus, a message exchange may look like: `socket.send("<msg>".encode()); sock.recv()`.

Pulsar is addressed via its default port number `pulsar://localhost:6650`. In Python, this works after `import pulsar` with `client = pulsar.Client(<url>)`, creating a first producer with a full topic URL `prod = client.create_producer("non-persistent://public/default/<topic>")` and then sending events with `prod.send("<msg>".encode())`. No acknowledgements need to be read. With `cons = client.subscribe(<topic-url>)` and `cons.receive(timeout_millis=50)`, events can be received in a non-blocking way. Finally, with the `pulsar-admin` command, functions can be deployed between input and output topic paths.

Due to the wide variety of systems, these are not further explored here but should be kept in mind when designing a data science architecture for streaming data. More information can be found in the documentation of ZeroMQ[17] and Pulsar[18].

Repetition

1. What is the main advantage of message brokers over polling approaches?
2. Why is the recv() method invocation necessary for ZMQ sockets?

6.5 Parallel and distributed computing

Parallel computing concepts were described before and command-oriented tools such as `parallel` were introduced earlier in this book as well. In this section, more capable data-oriented frameworks that combine automated parallelisation on one machine with distributed computation across several machines are introduced. These frameworks typically involve an initial overhead, but for larger quantities of data, their benefits become clear quickly. Among them are a local speedup by using multiple CPUs and even GPUs relative to the wall clock (at the expense of higher resource usage) as well as the enablement of processing of data volumes that no longer fit into the main memory of a single computer.

To make parallel computation available programmatically, several middleware systems and language-specific libraries exist. Among the more popular frameworks based on at least the map-reduce paradigm are Hadoop, Spark, Ray, Dask and Lithops. Further paradigms are supported by some of them with parallel and distributed processing capabilities, including composable pipelines, graph algorithms, windowed stream aggregation and data warehousing. The use of such a framework is not always required, especially when

[17] ZeroMQ: `https://learning-0mq-with-pyzmq.readthedocs.io/`
[18] Pulsar: `https://pulsar.apache.org/docs/`

software supports distributed operation internally. This is the case with some shells (`dsh` multiplexing) and compilers (`distcc`). In the following section, Spark is introduced from a Python (PySpark) perspective as exemplary framework for horizontal scaling of queries and user-defined functions in a compute cluster.

6.5.1 Data processing with Spark

Apache Spark[19] is a framework for parallel and distributed computing on sequences, tables and graphs as well as running relational database queries on tables. It offers interfaces for multiple programming languages, in particular Java, Scala and Python (PySpark). The execution model of Spark is driver-master-worker, with each worker covering one machine, each with potentially multiple CPU cores and GPUs. The application runs the driver that needs to be reachable from the workers, which limits deployment options. The master runs its main service on port 7077, to which the driver and workers connect, and a web interface on port 8080. Each worker runs a web interface on port 8081 to expose logs in addition to their main service port (e.g. 6666), and the driver additionally runs a web interface on port 4040. Furthermore, the driver opens a port to be reachable from the master and a second port for the block manager (e.g. 5555 and 4444, respectively). Many of those port numbers are only the starting points, as the driver automatically counts up in case a port number is already occupied. This complex handling of ports makes Spark not trivial to operate in a larger setting. Hence, in the following explanations, the focus is on the application interface for Python applications.

In a downloaded and extracted Spark folder, the interactive PySpark interpreter can be invoked with the command `bin/pyspark`. It can be used as a regular Python interpreter, but by embedding a Spark driver it offers access to the PySpark API through its pre-defined objects such as `sc` (Spark context), referring to a local in-memory worker, and `spark` (session object and SQL context). Alternatively, PySpark can be invoked as a wrapper around standalone applications (`spark-submit`) with the code importing the `pyspark` module to create the Spark context explicitly. This way, or alternatively by parameterising the PySpark interpreter, a Spark context to a remote cluster with potentially many worker nodes can be established with the line `sc = pyspark.SparkContext("spark://<sparkmaster>:7077", appName="...", conf=pyspark.SparkConf())`. This context then allocates the resources on the cluster until it is explicitly stopped with `sc.stop()`. Spark can also be embedded into other Python execution contexts such as scientific notebooks. For that purpose, instead of downloading the entire Spark release, one would typically run `pip install pyspark` in a virtual environment, along with op-

[19] Spark: `https://spark.apache.org/`

tional dependencies such as PyArrow and Pandas. A full download is however necessary to operate the cluster itself. If no cluster is available or needed, Spark can also parallelise computation within one computer (`local[*]`).

A number of options can be set on the `SparkConf` object as string key-value pairs. This includes `spark.driver.host` as driver hostname resolvable from the worker nodes along with the corresponding `spark.driver.port`, `spark.port.maxRetries` to raise the limit of concurrent connections over the default of 16 (e.g. 50 would be appropriate for a classroom setting), and `spark.cores.max` for the maximum number of requested cores. The assigned cores may be less if fewer are available, or temporarily zero (putting the application into waiting state) if none are available. Another option is `spark.jars.packages` to activate extension packages such as Graphframes for graph processing, Glow for genomics data, sparkMeasure to obtain metrics, and RAPIDS to get GPU acceleration.

Programming pipelines for data queries in Spark requires a good understanding of asynchronous programming concepts. Instructions are evaluated lazily, not necessarily at the line the instruction is written on, but rather when results have to be calculated. Usually this is at the end of pipelines or at explicit caching instructions in between. Moreover, despite being able to handle large amounts of data, Spark is not safe against out-of-memory situations. When performing an operation that joins data from all worker nodes or otherwise brings data to one place, the driver in PySpark may run out of memory.

Spark offers two main data structures, resilient distributed datasets (RDDs) for unstructured data and dataframes for structured, tabular data. A prerequisite is the setup of the context object as mentioned: `import pyspark; sc = ...`, for instance, with `spark-submit`: `sc = SparkContext(appName="<name>")`. An RDD can be produced programmatically with the command `rdd = sc.parallelize(<list>)` or (assuming shared storage access) by reading in a text file with `sc.textFile(<filename>)`. The distribution of the data partitions can be verified with the command `rdd.glom().collect()`. For structured data, the session/SQL context object must be set up first: `import pyspark.sql; spark = pyspark.sql.SparkSession.builder.getOrCreate()`. A dataframe can then be produced by adding a tabular schema to an RDD, by converting a Pandas dataframe, or by reading tabular data right away, such as in the command: `df = spark.read.format("csv").option("header", "true").load("file.csv")`. Relational SQL queries, graph processing and structured streaming are all implemented atop the dataframe API. Spark also provides a drop-in Pandas API for easier conversion of existing code towards distributed environments (`pyspark.pandas`). Further information about Spark programming is available from the respective guide.[20]

[20] Spark programming guide: `https://spark.apache.org/docs/latest/quick-start.html`

Repetition

1. What is the difference between using spark-submit and PySpark applications not making use of spark-submit?
2. The line `df = spark.read.format("csv").load("file.csv")` loads the indicated CSV file. Correct?

6.6 Model serving

Model serving refers to the provisioning and lifecycle management of trained machine-learning models, essentially statistical models, as a service. Such models contain read-optimised information about quantitative or categorical features found in the training data. The models allow inference and subsequent decision-making based on input data sent as a request to the service. The lifecycle management includes input data preparation, versioned training, testing and validation with the primary goal of achieving a high accuracy of predictions. Specific management tasks therefore encompass systematic model (re)training (split, test, validate, regression testing), version control (input data provenance, cleaning, lineage), delivery of model data and code (packaging, containerisation, registration, discovery, serving at scale) and operational concerns (authentication, logging, metering, monitoring). Access to such models for the purposes of inference and prediction should be possible from multiple applications, with high reliability and low latency. Accordingly, APIs for matching and prediction services should be generated automatically and integrate with workflow managers and message brokers. Among often-used model serving implementations are Iguazio, Kubeflow/KFserving, TensorFlow Serving/TFX, MLflow, Clipper and BentoML. The latter is briefly introduced here.

6.6.1 BentoML model serving

BentoML is a specialised middleware system to register machine-trained mathematical models and perform scalable inference on them. It comes with support for a number of ML libraries such as MLflow, TensorFlow, PyTorch, Keras, CatBoost, LightGBM, ONNX and Scikit-Learn. It is called a unified model serving framework due to this multi-format support. The models are transmitted as Python code via function call to BentoML which then physically stores them in a local directory in Pickle format. Hence, model persistence is achieved through files and directories, which can be optionally versioned by the use of Git. Moreover, BentoML allows for the automatic containerisation of models with Docker, with the containers starting a ready-to-use API for matching and inference. Moreover, it integrates with Airflow for model training pipelines or

workflows, specifically with Airflow's PythonOperator and PythonVirtualenvOperator as well as with Flink's stream processing capabilities for streaming model inference and with MLflow.

A common way to use BentoML is to create the predictor code in the form of a generic Python model class. The class should have a `__call__(inputlist)` method. Instantiated with input data, this becomes a predictor as Python model instance with data already loaded. This model can be persisted as BentoML saved model via `bentoml.pickable_model.save_model(<modelname>, <modelinstance()>)`, and executed with a runner. The runner may be implemented as follows:

```
import bentoml
# next line is only for testing
lmodel_content = bentoml.picklable_model.load_model("<
    modelname>:latest")
lmodel = bentoml.picklable_model.get("<modelname>:latest")
runner = lmodel.to_runner()
runner.init_local()
prediction = runner.run(<inputlist>)
print(prediction)
```

All saved models may be inspected in the shell with `bentoml models list`. For production use and access by multiple applications, custom service logic can then be defined in Python and delivered through an instance of the BentoML service. The implementation requires setting up a service object by instantiating `bentoml.Service(<service-id>, runners)` which then uses the `@<service-object>.api` decorator to specify input and output data formats. Again, a brief example is given:

```
import bentoml
import bentoml.io
runner = bentoml.picklable_model.get("<modelname>:latest").
    to_runner()
svc = bentoml.Service(<modelname>, runners=[runner])
@svc.api(input=bentoml.io.JSON(), output=bentoml.io.JSON())
def predict(inputlist):
    return runner.run(inputlist)
```

These services are launched through `bentoml serve <python-filename>:<service-object> --reload`. Tools like Curl can then be used to run predictions, for example: `curl -X POST -H "Content-Type: application/json" --data "[...]" http://127.0.0.1:3000/predict`.

BentoML services can also be exported as self-contained units with both algorithmic service logic and model data, through the command `bentoml build` based on instructions given in a service description file `bentofile.yaml`. A minimal service description would be `service: "<modelname>:svc"`. All

such built units can be inspected with `bentoml list`, and exported for model transfer with `bentoml export <modelname>:latest <modelname>.bento`.

These units can in turn then be converted into Docker container images via `bentoml containerize <modelname>:latest`, based on additional `docker` entries in the YAML file specifying the base image and Python interpreter version. Containers then serve the model and allow for using it (e.g. for predictions) on port 3000. Hence, they can be run individually with the command `docker run -it -rm -p 3000:3000 <modelname>:<tag> serve`.

A step-by-step documentation with further information is available on the BentoML website.[21]

Repetition

1. Why are there dedicated model serving implementations when instead a web server could be used for distributed access to a model?
2. Why does Bento need a POST request for the prediction? Predictions are read-only and thus should work with a GET request.

6.7 Data integration

Data integration refers to the ability to shuffle data between data stores, or between applications, while performing conversion and transformation. Format conversion would, for instance, take CSV input from a source and deliver it as JSON to a sink. Transformations happen within the same format and would filter, aggregate or augment the input data. There are many approaches to data integration referred to as ETL or ELT, a reference to the order of steps in Extract-Transform-Load integration processes. Recent implementations to set up generic integrations include Meltano, Arrow Flight and dbt-core.

6.7.1 Meltano data integration

Meltano aids in setting up pipelines for large-scale data transformations from sources to destinations. It is installed through pipx with the command `pipx install meltano` and can then be activated for a project directory via `meltano init <projectname>`. For the affected directory, Meltano creates a number of default files and empty directories and administers several runtime environments such as development, staging and production. The main configuration file is `meltano.yaml`, which has the default environment set to development (`dev`). The project directory is designed in a way that it can be curated in Git, so that an additional `git init` is recommended. In their initial state, all

[21] Bento documentation: `https://docs.bentoml.org/en/latest/concepts/model.html`

environments are empty and waiting to be configured with a sink-source combination or, in Meltano terms, either extractor-loader or tap-target combination. The configuration may additionally encompass transforms and transformers, files, utilities and other pipeline ingredients.

The commands `meltano discover extractors` and `meltano discover loaders` are used to define the data transformation pipeline. There are more than 500 extractors available, alls prefixed with `tap`, and more than 30 loaders, all prefixed with `target`. Many exist in multiple implementation variants, so that, in practice, extracting data from a system or loading it into another system may only work well with specific variants. For the purpose of testing and interoperability, there are some generic loaders such as JSON files (`target-jsonl`).

The extractor-loader combination is installed into the current Meltano environment, but not yet configured, with `meltano add <extractor/loader> [--variant=<variant>]`. This command adds the chosen extractor or loader to `meltano.yaml` but also places its implementation into the `plugins` directory. Subsequently, a mandatory configuration is conducted with `meltano config <extractor/loader> set --interactive`, which asks the user about few to many key-value settings such as endpoints, credentials and options. Moreover, `meltano select <extractor> --list --all` followed by `meltano select <extractor> <table> <column>` selects a source table (entity) and column (attribute) in case multiple are offered by the extractor.

Finally, `meltano run` runs the pipeline that in turn extracts data from the configured source and sends and loads transformed data into the configured destination. While the run command runs once, a regular run to catch updates to the data source can also be configured with the `meltano schedule` command and activated with `meltano invoke`, both taking additional parameters. Schedules in particular can be set up with commands of the form `meltano schedule add <schedname> --extractor <extractor> --loader <loader> --transform [run|skip|only] --interval "@hourly"`. Many commands are predefined and others can be added with some configuration, including Jupyter notebooks as steps within a pipeline. To maintain an overview about pipelines, a web-based user interface is also available through the blocking command `meltano ui` for subsequent access at `http://localhost:5000/`.

A detailed documentation is available online.[22] Hundreds of plugins, specifically for different data sources, are available from the hub.[23] A managed service for Meltano is not yet available but expected to arrive sometime in the near future. There are also more complex alternatives to Meltano such as Pachyderm that require a more sophisticated deployment and configuration.[24]

[22] Meltano documentation: `https://docs.meltano.com/getting-started`
[23] Meltano hub: `https://hub.meltano.com/`
[24] Pachyderm: `https://www.pachyderm.com/`

Repetition

1. There are targets, loaders, extractors and taps. Which of these terms are practically synonyms?
2. What is the advantage of maintaining the data integration configuration in a Git repository?

6.8 Workflows and distributed scheduling

Meltano and other data integration tools represent static workflows with a source, a sink and a set of transformation rules. In many scenarios, more complex workflows with tree and graph structures need to be set up and triggered through various events including matching times. Scheduled task invocation beyond the use of Cron is broadly supported in programming environments, for instance, through Celery or, with emphasis on parallelisms similar to Spark, through Dask. Such workflows may however also occur in continuous delivery scenarios triggered by a change in data or code, for instance, via Git hooks and corresponding workflows like Gitlab Pipelines. The complexity increases in workflow languages permitting subtasks, branching, looping, nesting and other internal control structures. Corresponding workflow systems need to support automation (scripting and APIs), robustness (checkpointing, restarts) and optimisation (caching, adaptive re-entrant execution). They also need to support regulated environments through logging as well as lifecycle and user management. Finally, they should be flexible concerning the task implementation: as shell commands, Python functions, API calls, ETL processes and others. In the following, Airflow is introduced as a representative system for both scheduled invocations and workflows, supporting many of the listed requirements.

6.8.1 Airflow task and workflow specification

Apache Airflow[25] is a workflow engine, a scheduler and a programming interface to express complex executable workflows. Workflows are expressed as DAGs in Python in three main parts: metadata, a set of tasks and an execution order specification for those tasks. A task is triggered by a timer event, by a previous task having finished, or by a data dependency having changed. For that purpose, Airflow makes available an `airflow` Python module, and a script file importing this module and making use of it represents a workflow. The package with module and executable can be installed into a virtual environment with `pip install airflow`. The service start command `airflow`

[25] Airflow website: `https://airflow.apache.org/`

`standalone` listens on two ports: 8793 for internal scheduling purposes and 8080 for the web interface. The following listing shows a customised and decomposed start sequence.

```
airflow db init
airflow users create \
--username admin \
--firstname X \
--lastname Y \
--role Admin \
--email Z # enter password, or use --password
airflow webserver --port 8080
airflow scheduler
```

With `airflow dags list --subdir .`, all DAGs present in the current working directory are summarised. If needed, calling `airflow dags test --subdir . <dag_id>` tests the execution of a specific workflow.

The `DAG` class from the `airflow` module needs to be instantiated with a unique identifier given as `dag_id`. A particular design decision of Airflow is that workflows are addressed later by this identifier, independently of the name of the containing Python file. Further parameters refer to the workflow ownership and its timing information, including start date and interval in Quartz cron syntax. Hence, a typical Airflow specification may start as follows:

```
from datetime import timedelta
import datetime
import airflow
import airflow.utils.dates

with airflow.DAG(
  dag_id="helloworld_bash",
  default_args={"owner": "airflow"},
  schedule_interval="0 0 * * *",
  start_date=airflow.utils.dates.days_ago(2),
  dagrun_timeout=datetime.timedelta(minutes=60),
) as dag:
  ...
```

Tasks are specified as instances of operator classes along with a unique identifier and operator-specific parameters. The identifier is a string given as `task_id` parameter. The instances can be named differently but for simple cases may follow a convention of `t1`, `t2` and so forth. The following operator classes are among the often used ones:

1. EmptyOperator. This operator does nothing (no-op) and has no further parameters. It can be used as placeholder in a workflow, similar to a `pass` instruction in Python.

2. PythonOperator. Executes a Python function that must be reachable, either within the same file or from an imported module. The function is referenced with the `python_callable` attribute, and is completed by the keywords argument `op_kwargs` to specify parameters that should be passed to the invoked function. Alternatively, the Python operator can directly be specified as decorator of an existing function, like this: `@task(task_id="...")`.

3. PythonVirtualenvOperator. A variant that supports custom module deployment into virtual environments.

4. BranchPythonOperator. This special operator makes it possible to choose one out of several branches by evaluating a condition.

5. BashOperator. This operator works similar to `os.system()` by synchronously executing one shell command. The command is passed as `bash_command` parameter.

6. PapermillOperator. Loads a Jupyter notebook from the absolute path in `input_nb` and, since such notebooks are modified during execution, saves the result notebook as `output_nb`. Optionally, a dictionary `parameters` may be passed to parameterise the notebook according to the Papermill conventions, which are documented in the Jupyter notebook section of the book.

Further operators exist for system and data integration, including SimpleHttpOperator, EmailOperator, MysqlOperator and PostgresOperator. With those operators, Airflow overlaps functionally with Meltano, although both can also be integrated with `meltano add orchestrator airflow` followed by `meltano invoke airflow scheduler` and `meltano invoke airflow dags list`. The invocation of operators follows tristate semantics: 0 signals success, 99 signals skipping, and all other numeric codes signal failure.

The last line of a simple Airflow file specifies the order of execution and therefore constructs the actual workflow. Two greater-than signs are used to create a sequence, e.g. `t1 >> t2`. Tasks that can be parallelised due to not depending on each other's results are grouped in square brackets, e.g. `[t3, t4]`.

Repetition

1. A BashOperator t1 creates a file, and a PythonOperator t2 reads that file. Would [t1, t2] be a valid workflow?
2. What does the schedule interval used in the example (* * 0 0 0) mean?

Chapter 7

Collaboration and Governance Platforms

In contrast to the last two sections that focused on locally run shell tools and locally operated middleware, this section dives deeper into provisioned and managed middleware online platforms to accomplish data science-related governance and collaboration tasks. These platforms typically offer a web interface and, while they can be operated locally for a single user, have the capability to serve multiple users in teams and thus facilitate collaboration supported by extensive authentication and authorisation schemes. Consequently, they are often consumed as pre-provisioned setups and managed services supporting the data science workflows on a team or institution basis with oversight, change management and compliance concerns.

Three exemplary collaboration platforms with suitable open source licencing are introduced: Jupyter, Gitlab and Open Data Discovery. Data scientists should be familiar with the main workflows within these platforms and are then able to also use similar platforms. Such alternatives are mentioned in each respective section. Where applicable, possibilities for integration of the previously mentioned tools and middleware are highlighted.

7.1 Scientific notebooks

A digital scientific notebook is a web-based environment subdivided into input cells that either contain static information or run dynamic code, such as Python, R or Julia. Output cells then contain text, plots and other content. In contrast to a script, the cells can be run selectively. This might speed up explorative data analysis, but it might also lead to inconsistencies when the dependency order between cells is not manually considered. Moreover, there

is no explicit termination unless the code execution context (kernel) is terminated. This means allocated resources may be blocked unless explicitly freed up programmatically or implicitly by kernel termination.

Various notebook implementations with single-language or polyglot kernels exist, such as Jupyter which is described next, Apache Zeppelin[1], Querybook[2], and Polynote[3].

7.1.1 Jupyter notebooks

Jupyter is a popular implementation of notebooks that can run in isolation, but also shared among team members. In Python Jupyter notebooks, the kernel can be chosen; by default, the iPython interpreter is used with support for special commands starting with `!` (system execution) or `#`. Big data frameworks such as PySpark can be integrated into notebooks through the corresponding Python modules to perform parallel computing tasks by offloading them to a larger cluster. There are also Jupyter extensions such as Toree and BeakerX for Spark, and Elyra to visually design workflows that can then be mapped to Airflow or Kubeflow implementations.

Jupyter can be operated as single instance, catering to a single user or multiple users in a fully shared environment without isolation. It can also run as JupyterLab, combining the notebook with web-based text editors, terminals, launchers and custom widgets, while still catering to single users. For multi-user environments with isolation between users, JupyterHub can launch containerised notebooks, including on container orchestrators such as Kubernetes.

7.1.2 Working with notebooks

Launching an interactive single-user or fully shared notebook environment locally is achieved with the subcommand `jupyter notebook` or the alias command `jupyter-notebook` (with dash). The command hangs to serve a web browser at the URL containing a secret token indicated at standard output, and can be terminated with the keyboard combination Ctrl+C. The token can be disabled by running `jupyter notebook password` and thus setting a password permanently. Headless mode can subsequently be used by adding `--no-browser`. The default port number is 8888 and can be changed with the parameter `--port`. Furthermore, notebooks only listen to local connections by default. To be network-enabled, the parameter `--ip 0.0.0.0` needs to be set. The notebook serves the entire current working directory available. One

[1] Zeppelin website: `https://zeppelin.apache.org/`
[2] Querybook website: `https://www.querybook.org/`
[3] Polynote website: `https://polynote.org/latest/`

further important parameter is therefore `--notebook-dir <dir>` to confine the access to a certain directory.

A number of configuration settings becomes available through files only. This includes the ability for users to switch off the running Jupyter instance with a Quit button. In the file `~/.jupyter/jupyter_notebook_config.json`, the setting `"quit_button": false` removes this ability. JavaScript-based customisations such as default contents in new notebook can furthermore be performed by placing appropriate files into the directory `/usr/share/jupyter/nbextensions/` and activating the extension with a JSON configuration file in `/etc/jupyter/nbconfig/notebook.d/`.

The batch execution of notebooks including parameterisation is the task of Papermill[4], which can be installed with `pip install papermill` and requires adding `~/.local/bin` to `$PATH`.

Notebook files are stored with the extension `.ipynb` (from the original name iPython notebook) and can also be exported as Python scripts to facilitate automation.

Jupyter notebooks can be tried out for free[5] although that is not recommended for important scripts or confidential data. A more convenient service is Binder to execute notebooks already stored on the Internet in a Git repository.[6]

7.2 Code and data lifecycle management

The management and easy use of versioned repositories, possibly backed by Git or other version control system, is the domain of online repository frontends. They provide functionality for creating hierarchies of projects and subprojects with multiple repositories, assigning access rights, web-based read access to existing files and even web-based write access for the creation of new files. Such collaborative platforms also allow for planning and discussing the content in terms of code and data, for forking variants and for submitting wishlist and bug reports. Moreover, they support actions based on changes to the code or data, such as the automated rebuilding of binary packages.

Apart from shell-oriented tools such as Gitolite, and the built-in Gitweb as well as cgit, there are many web-based Git management platforms. Gitlab is one of them and is presented next. Potentially leaner alternatives include Gogs[7], Gitea[8] and Trac[9].

[4] Papermill documentation: `https://papermill.readthedocs.io/`

[5] Jupyter demonstration: `https://jupyter.org/try-jupyter/retro/notebooks/?path=notebooks/Intro.ipynb`

[6] MyBinder: `https://mybinder.org/`

[7] Gogs website: `https://gogs.io/`

[8] Gitea website: `https://gitea.io/en-us/`

[9] Trac website: `https://trac.edgewall.org/`

7.2.1 Gitlab as repository management platform

Gitlab is a collaborative environment around a fully managed set of Git repositories. The most obvious functionality of the platform is the creation of personal projects and project groups and the definition of Git repositories within these projects. On the collaboration side, it allows sending invitations to users who are added with differentiated access rights into the projects depending on the chosen role. Full access rights are available to owners. Creating a new project results in ownership automatically. Fewer access rights are available for maintainers, developers, reporters and finally guests. For instance, only an owner can delete a project, and only owners and maintainers can rename a project or change its settings.

The platform supports patch management in the form of pull requests for those not sufficiently privileged or unsure about direct file modification in one of the Git branches. Moreover, it supports the definition of server-side hooks that run whenever a commit is pushed. This feature can be used to implement a continuous data processing pipeline, for instance, updated training whenever new data arrives.

Finally, Gitlab also contains registries for packages (e.g. Python packages similar to PyPI) and container images. The Docker container registry runs on port 5050. The command `docker login <gitlabserver>:5050` configures the local Docker client to make use of it. Images can then be pushed to `<gitlabserver>:5050/<user>/<project>/<image>`.

Due to many system integrations and complex configuration, the native installation of Gitlab requires elevated privileges and several required post-installation configuration steps. The installation process is described in the documentation.[10] A more portable approach is to run Gitlab as a container, in the following form:

```
export GITLAB_HOME=$HOME/gitlab
sudo docker run -ti \
  --hostname localhost \
  --publish 30000:80 \ # Container port 80 becomes 30000 on
      the host
  --name gitlab \
  --restart always \
  --volume $GITLAB_HOME/config:/etc/gitlab \
  --volume $GITLAB_HOME/logs:/var/log/gitlab \
  --volume $GITLAB_HOME/data:/var/opt/gitlab \
  --shm-size 256m \
  gitlab/gitlab-ee:latest
```

[10] Gitlab documentation: `https://about.gitlab.com/install/#ubuntu`

For security reasons, a password is auto-generated upon the first invocation. The additional command `sudo docker exec -it gitlab grep 'Password:' /etc/gitlab/initial_root_password` retrieves the password from the container and displays it on standard output.

Gitlab as a managed service[11] is similar to Github, with the added advantage of being open source and thus more fit for self-hosting. The Gitlab CLI can be used to interact with Gitlab instances. An exemplary invocation is to list all active projects with `python-gitlab --server-url https://<gitlabserver> project list`.

7.2.2 Gitlab as delivery platform

Based on any changes to data, code or other managed artefacts, Gitlab can run processes for continuous integration (CI) or continuous delivery (CD). Both terms largely overlap, but the former typically refers to a build–test–release process, whereas the latter refers to the deployment of artefacts to production systems, ready to be delivered to users.

To define what should happen upon a change, Gitlab uses the concept of Git hooks and connects them to pipelines and web hooks. Web hooks are external APIs that get called on certain events such as repository pushes or changes in the issue tracker.

Pipelines executing internally in Github are also represented by a webhook mechanism. The web hook endpoint then follows the form `https://<gitlabserver>/api/v4/projects/27/ref/REF_NAME/trigger/pipeline?token=TKN`. Despite the name, pipelines can be not only basic pipelines in the form of sequences but also DAG workflows representing full workflows or even multi-project pipelines to automate a larger release process. Each pipeline works in defined stages such as build, test or release as well as jobs representing steps of each pipeline. The pipeline definition is edited as a YAML file and can be compared to a makefile in terms of providing targets that trigger command sequences, with each command being a job. The following listing gives an example of a two-stage pipeline sketch:

```
stages:
  - prepare
  - test
image: alpine # global or per job; must contain job scripts
prep_a:
  stage: prepare
  script: # before_script + after_script also available
    - echo "This job prepares something."
    - wget ...
```

[11] Gitlab.com service: `https://about.gitlab.com/`

```
prep_b:
  stage: prepare
  script:
    - echo "This job prepares something else."
    - jupyter-execute ...
test_a:
  stage: test
  script:
    - echo "This job tests something. It will only run when
        all jobs in the"
    - echo "preparation stage are complete."
```

Gitlab runners execute the jobs, for instance, in the form of containerised tools to provide isolation. This way, Gitlab can also execute an Airflow image so that a more sophisticated workflow is initiated under the control of Airflow. Gitlab metadata can also be pulled by Meltano, by using the `tap-gitlab` extractor based on a configured access token in the project's settings menu next to the webhook configuration.

The global Gitlab registry contains re-usable pipeline components such as linters that can be used to rapidly construct useful pipelines for production scenarios.

7.3 Data catalogues and governance

There are two main flavours of these platforms. Some are more focused on publishing single datasets, often file-based, along with metadata. Implementations include CKAN[12] and Magda[13]. Others are more focused on ETL processes, data provenance and lineage. Those include many emerging platforms that are often still non-trivial to deploy, such as Open-Metadata[14], Datahub[15], Amundsen[16] and Apache Atlas[17]. The Open Data Discovery platform (ODD) is also in this camp and are briefly introduced next.

7.3.1 ODD deployment

ODD[18] can be up and running with moderate effort when using the provided container composition. First, the code repository needs to be cloned to get access to all relevant launch files (size ca. 50 MB). The referenced container

[12]CKAN website: `https://ckan.org/`
[13]Magda website: `https://magda.io/`
[14]Open-Metadata website: `https://open-metadata.org/`
[15]Datahub website: `https://datahubproject.io/`
[16]Amundsen website: `https://www.amundsen.io/`
[17]Atlas website: `https://atlas.apache.org/#/`
[18]ODD website: `https://opendatadiscovery.org/`

images are then automatically downloaded (size ca. 900 MB). The instructions are as follows:

```
git clone https://github.com/opendatadiscovery/odd-platform
cd odd-platform/
docker compose -f docker/demo.yaml up -d odd-platform-
    enricher
# or docker-compose in older Docker environments
```

The web interface to the platform is then running on port 8080, whereas the platform's PostgreSQL database can be accessed on port 5432. If the setup happens on a remote virtual machine, the ability to use SSH port forwarding becomes handy again: `ssh -L 10080:localhost:8080 ubuntu@<vmhost>`.

Data can be pulled from many sources through collectors including files, relational tables and message broker topics. MySQL, PostgreSQL, MongoDB, Airflow and Kafka are among the supported systems. In addition to input data, ODD supports transformers, quality checkers and other typical data integration pipeline elements.

Custom collectors can be added as well. In the management interface, one would click on 'add collector' and choose an arbitrary name for it. An access token is generated and can be copied from the interface for pasting into the collector configuration file. For demonstration purposes, one such file for a custom PostgreSQL connector is already provided in the code repository: `docker/config/collector_config.yaml`. With the token pasted into the empty token string within this file, the command `docker compose -f docker/demo.yaml up -d odd-collector` pulls additional Docker images (size ca. 2 GB) and adds the custom data source. In the web interface's catalog menu, filtering can be used to focus only on this source (named `postgresql-step2-test` by default) to show all datasets and other pipeline elements available from this source. For curation purposes, string tags and key-value metadata entries can be added to each element.

Repetition

1. How does the process hierarchy look like if the command os.system("ls") is invoked from a Jupyter notebook?
2. What is the command for pulling Docker images from a Gitlab container registry?

Chapter 8

Execution and Orchestration Platforms

Most of the software covered in this book is deployable in the form of system-wide packages per-user packages, or container images. When software becomes more complex, it might require a lower-level configuration of storage and networking resources, autoscaling rules, proxies and orchestration logic. The execution of containers or of virtual machines under this configuration is then controlled by dedicated platforms. In this chapter, two representative platforms for the execution of virtual machines and containers are briefly introduced: OpenStack and Kubernetes. Moreover, a mapping of those platforms and the previously discussed middleware and collaboration platforms into cloud services is given. The coverage is not meant to be an extensive guide into orchestration but rather to convey sufficient knowledge to be able to deploy data science tools should they require such a setup.

8.1 Virtual machines management

Virtual machines are highly isolated execution contexts on top of abstracted hardware, including both kernel and userland applications in a guest context on top of the host operating system. A hypervisor configures the available hardware resources and ensures that all guest system calls are appropriately translated into host calls. Hardware-supported virtualisation is often available for this task. On Linux, the Kernel Virtual Machine (KVM) has become the dominant hypervisor. However, this still requires managing the virtual machine images that should be executed, along with their configuration known as instance types. For example, a machine learning application might require 4 virtual CPUs (vCPUs), 1 GPU, 8 GB of memory, 100 GB of disk

space, and a public IP address. Tools like `virsh`, `virt-install` or plain `kvm`/`qemu-system-x86_64` are powerful but not easy to operate.

Instead of interacting with the hypervisor directly, setting up all configuration is eased by web-based infrastructure management software. In the next section, OpenStack is introduced. Possible alternatives include Apache CloudStack[1], OpenNebula[2] and Proxmox[3]. Eucalyptus[4] has been one of the first platforms in this space but might be no longer actively maintained. Similarly, Danube Cloud[5] might be a candidate platform to look at with the same caution.

8.1.1 Using OpenStack web interface and API

OpenStack[6] is a set of named components to manage infrastructure, primarily virtual machines and containers. Not all components need to be active, and therefore OpenStack instances differ in their functionality. A typical component combination might be Glance to manage virtual machine images, Nova for interaction with the hypervisor running virtual machine instances, Swift or Cinder for object or block storage, respectively, Keystone for identity management and Heat for orchestration.

All services integrate into Horizon, the web-based management interface. Users log into this interface, see their assigned projects with quotas, and set up virtual machines and other resources like virtual block devices within the project. Accounts, instance types, quotas, projects and other privileged configuration settings are handled by a system administrator. Hence, to get started with an instance that is not self-operated, a user would first have to request access from the administrator, by specifying the desired resources, and would subsequently get the account and the ability to log into Horizon.

Inside the interface, users would register SSH keys, network policies and, optionally, secondary persistent block storage devices. Then, they would launch VMs, each of which comes with a primary block storage device of reasonable but perhaps limited capacity. A typical instance type might have 4 vCPUs, 1 GPU, 8 GB of memory and 40 GB disk space. Then for the aforementioned machine learning application, a secondary block device is necessary. Within the Horizon dashboard, the block device would receive a label, which can then be used to automount it inside the VM by putting a line like `LABEL=seconddisk /home/ubuntu/mountpoint ext4 defaults 0 0` into the file `/etc/fstab`.

[1] CloudStack website: `https://cloudstack.apache.org/`
[2] OpenNebula website: `https://opennebula.io/`
[3] Proxmox website: `https://www.proxmox.com/en/proxmox-ve`
[4] Eucalyptus website: `https://www.eucalyptus.cloud/`
[5] Danube Cloud website: `https://danube.cloud/`
[6] OpenStack website: `https://www.openstack.org/`

By default, VMs are only accessible on an internal network accessible by all VMs, with IP addresses of the form `10.0.x.y`. OpenStack has the concept of floating IP addresses which depending on the quota settings can be assigned to running VMs to make network services on them accessible from outside. A VM then possesses three network interfaces that can be distinguished to enforce access policies: localhost, the internal IP, and the floating IP.

Access to VMs can be granted to users who do not have an OpenStack account, by requesting the desired resource configuration from them, as well as the SSH public key, and setting the VM up for them. The owner of a virtual machine, possibly a data scientist, then merely has the tasks of maintaining the VM itself, including the regular installation of security updates, and ensuring that no unnecessary services are exposed to the world. Regular maintenance also involves checking for resource shortage using `df`, `free` and similar tools introduced previously in this book.

Access to OpenStack is also possible programmatically. The lifecycle management of virtual machines serves as an example. Due to such VMs being managed by Nova, the first step is importing the Python submodule `novaclient.client` and setting up a client object. Next, the list of virtual machines is retrieved. Each machine has a current state and, in case a task such as powering on or off a VM is still running, also a task state. The following code exemplifies starting all switched off VMs inside a project, unless they are already in the process of being started. The authentication URL follows the pattern `https://<openstack-domain>:5000`.

```
import novaclient.client

nova_client = novaclient.client.Client(version="2",
    username=os.getenv("OS_USERNAME"), password=os.getenv("
    OS_PASSWORD"), project_name=os.getenv("OS_PROJECT"),
    auth_url=os.getenv("OS_AUTH_URL"), user_domain_name="
    default", project_domain_name="default")
servers = nova_client.servers.list()
for server in servers:
  status = server._info["status"]
  taskstate = server._info["OS-EXT-STS:task_state"]
  if status == "SHUTOFF" and taskstate is None:
    print("Starting machine", server.name)
    server.start()
```

A nested dictionary of network interfaces is also available through the object attribute `server.addresses`. Each inner dictionary can be checked for the presence of a fixed IP address assignment, for instance, with: `ip["OS-EXT-IPS:type"] == "fixed"`. The field `ip["addr"]` then contains the IP address.

8.2 Container management

A lot of software is shipping with the option to run in containerised form, or even requires container-native deployments. Container orchestration platforms ensure that they can run in production with the right provisioning and scaling characteristics. Over the past years, Kubernetes has emerged as one of the dominant orchestrators, with many features but also correspondingly high complexity. It is briefly introduced in the next sections.

8.2.1 Kubernetes ecosystem

Kubernetes runs as a standalone system or more typically as a cluster involving multiple virtual or physical machines. Within a running instance, different namespaces can be set up to increase isolation. By default, the namespace `kube-system` is reserved for Kubernetes-internal resources.

Kubernetes orchestrates resources of different types, each with corresponding declarative configuration. YAML or JSON files describe resources such as containers instantiated from images (with the types `Deployment`, `StatefulSet`, `Job`/`CronJob` and others), exposed network interfaces (`Service`) and storage areas (`PersistentVolumeClaim`). Many of these resources have a physical representation, such as CPU and memory limit assignments in deployments. Custom resources such as bindings to other platforms can be developed and deployed. Operators are further containerised software components that automate the lifecycle.

Multiple implementations of Kubernetes exist, especially to account for smaller (single-node) operation when horizontal scaling is not required. Among the well-maintained implementations are Minikube[7], K3s[8], MicroK8s[9] and Kind[10]. They differ especially in terms of how easily extensions can be installed and how strictly authentication needs to be performed. There are also multiple platforms building on top of Kubernetes, especially with additional features for application engineers, such as OpenShift and CloudFoundry.

The command-line utility `kubectl` is used to interact with a Kubernetes cluster. It works based on contexts, so that for each cluster a specific context can be set up and working across clusters becomes possible. With this tool, applications can be deployed and configured and the cluster status can be monitored. With OpenShift, the `oc` tool can be largely used as a substitute. Similarly, with Minikube, K3s and MicroK8s, bundled wrapper commands exist (e.g. `k3s kubectl`) so that a separate installation is not necessary.

[7] Minikube website: `https://minikube.sigs.k8s.io/docs/`
[8] K3s website: `https://k3s.io/`
[9] MicroK8s website: `https://microk8s.io/`
[10] Kind website: `https://kind.sigs.k8s.io/`

Helm[11] is a package manager for Kubernetes applications. On the shell, the tool `helm` allows for updating information about packages and installing them to a cluster namespace. These packages, also called Helm charts, are in essence templated resource descriptions along with metadata and dependencies, allowing for a higher-level handling of more complex applications. The Artifact Hub[12] is a popular repository for such applications. There are also alternative tools to adjust applications to specific needs, for instance Kustomize.

8.2.2 Kubernetes installation

In the following, MicroK8s is used as exemplary implementation to obtain a self-hosted Kubernetes instance. On an Ubuntu system, the installation of Kubernetes servers is performed with a dedicated meta-package: `sudo apt-get install kubernetes` followed by `kubernetes install` and choosing option 1, MicroK8s. The metrics server can be enabled as extension with `microk8s enable metrics-server`, and then works out of the box without requiring authentication. With `alias kubectl="microk8s kubectl"`, client-side access to the Kubernetes instance becomes possible.

In case a Kubernetes instance is already provisioned elsewhere, an alternative path needs to be taken. First, the instructions to download it need to be followed[13], and, second, an access context needs to be configured. The following listing shows an exemplary use:

```
# packaged: if available, use that and skip the three
    manual lines below
sudo apt-get install kubernetes-client
# manual: get current version first, and use it instead of
    1.27.2 below
curl -LS https://dl.k8s.io/release/stable.txt
curl -LO https://dl.k8s.io/release/v1.27.2/bin/linux/amd64/
    kubectl
sudo install kubectl /usr/local/bin/kubectl

kubectl config set-cluster <clustername> --server=<host>
kubectl config get-clusters
```

As a first confirmation of a successful deployment, `kubectl get all` should show all resources in the `default` namespace. In particular, there should be a Kubernetes service (`service/kubernetes`) with a cluster IP address. Other namespaces can be chosen, for instance by appending `--namespace kube-system`. To get a list of all namespaces, `kubectl get namespaces` is

[11] Helm website: `https://helm.sh/`
[12] Hub website: `https://artifacthub.io/`
[13] Kubectl guide: `https://kubernetes.io/de/docs/tasks/tools/install-kubectl/`

helpful. With `kubectl top nodes`, statistics about the resource consumption can be shown, and with `kubectl api-resources`, all resource types known to the instance are shown as well. Access to internal information can be achieved by combining the tool with post-processing, as in: `kubectl get service --namespace kube-system -o json | jq ".items[0].spec.clusterIP"`.

When these commands work as expected, the Kubernetes instance can be used to deploy applications.

8.2.3 Working with Kubernetes

Assuming a working Kubernetes instance and properly installed tools (`kubectl` and `helm`), an application can be installed in two ways. If it is packaged for Helm, the steps follow the sequence of commands listed below, assuming a self-chosen release name:

```
kubectl create namespace <namespace>
helm repo add <name> <repo-url> # for custom applications
    not in Artifact Hub
helm repo update
helm search repo <searchterm> # or 'hub' for Artifact Hub;
    lists charts
helm install <release-name> <chart> # simple test
    installation; more complete:
helm upgrade --namespace <namespace> --install <release-
    name> <chart>
```

Otherwise, if only YAML files are provided, a typical sequence would look as follows:

```
kubectl create namespace <namespace>
kubectl apply -f <app>.yaml --namespace=<namespace>
kubectl get deployments # verify that deployments show up
```

Static scaling of containers can be set up directly as configuration instructions on the container-related resources such as `Deployment`. For vertical and horizontal autoscaling, more sophisticated resources exist, including `VerticalPodAutoscaler` (VPA). Whether or not a certain autoscaling mechanism is offered in a Kubernetes instance can be found out by querying the available APIs with `kubectl get --raw /apis`. It contains the namespaced list of API groups. Those of relevance to autoscaling are `autoscaling/hpa` for horizontal scaling, `autoscaling.k8s.io/vpa` for vertical scaling and finally `keda.sh/keda` for event-driven scaling. Most scaling mechanisms (static, VPA and HPA) are reactive based on CPU and memory consumption, whereas KEDA can also react on network traffic and other events. In contrast to the

others, VPA is able to calculate recommendations on predicted resource requirements. Usually, the autoscalers are used in combination with the metrics server, `metrics.k8s.io/metrics`.

```
# configure HPA autoscaling
kubectl autoscale deployment <deployment> --cpu-percent=50
    --min=1 --max=3
# verify HPA autoscaling status
kubectl get hpa -o json | jq ".items[0].status"
```

8.3 Cloud services

Cloud computing refers to the on-demand provisioning of programmable services representing infrastructure, platforms, applications and data. Application orchestration in commercially offered clouds combines execution in virtual machines, containers and other environments with managed middleware services. Long-running virtual machines and containers are typically offered as Infrastructure-as-a-Service (IaaS), containers also as Container-as-a-Service (CaaS) hosting. Short-running containers and function implementations are also offered as runtime for Function-as-a-Service (FaaS). Likewise, databases may be offered as DBaaS, and other middleware as Backend-as-a-Service (BaaS).

Many cloud providers run Kubernetes for CaaS, along with KNative for scale-to-zero containers as backing for FaaS. Moreover, some providers also run registries for VM and container images. Hence, working with a cloud is not difficult when the fundamental technologies and platforms are understood, primarily by mapping the cloud service product names to the names of the platform and infrastructure technologies. Sometimes, cloud providers are explicit about what technology is used under the hood, and sometimes this can be found out by trying. Nevertheless, some cloud services also run proprietary software not available outside of the provider, and some run heavily modified or customised software. In the interest of interoperability and avoidance of vendor lock-in, the subscription to such services should be assessed carefully.

Repetition

1. Is a virtual machine managed in OpenStack aware of its public IP address?
2. Does Kubernetes support proactive autoscaling?

Chapter 9

Global Infrastructure

Beyond the individual hosted tools and online platforms, some online platforms offer much more combined functionality and thus qualify as complete self-service infrastructure for data scientists. While it may be technically possible to operate them locally, their complexity no longer permits doing so in an affordable manner. This holds for platforms located in a single region and even more so for truly globally distributed infrastructure such as content delivery networks (CDNs) and edge clouds, serving their users in geographic proximity. To give a second reason, the reliance of the platforms on social interaction workflows often stimulates a single global instance instead of isolated personal or institution-internal instances. Similarly, having a single global entrypoint facilitates discovery and bootstrapping, and a single global data broker for non-confidential data facilitates integration. Hence, the selection of appropriate platforms, account creation, subscription management and reliance on internal administrators or external operators needs to be factored into the equation when designing automation for data science.

In general, there are two paths towards global platforms: federation and dominance. To compare, consider the case of online social networks: In many parts of the world, Twitter had become the dominant commercial platform for broadcasting short messages. Mastodon, XMPP and other implementations allow for federation and therefore decentralised control of the hosting but, lacking the budget power, would not cause the same necessary network effect. In data science, few platforms support federation, and therefore global infrastructure is primarily reliant on dominant players, both full-stack cloud/CDN providers and providers of other platforms. Examplary platforms with lock-in risks include Github, social networks and hyperscalers. While most of those platforms are commercially operated based on mandatory subscriptions, there are always long-lasting global infrastructure providers with free tiers or even entirely free offerings that are especially suitable for learning the technology.

In the following, two worthwhile global infrastructure platforms based on open source implementations are described: Renku Lab and Zenodo. They are complemented with a national infrastructure service on the data input side: OpenTransportData. Together, they can be used to form a whole data pipeline without self-operated infrastructure. Complementary services for discovery (Etcd) and streaming (Dweet) are also described to facilitate distributed applications with various data processing and analytics components. One commonality is that these services and platforms are available for free and invite studying their functionality without having to put a credit card on file. They are also fairly easy to get started with and automate without unnecessary obstacles such as two-factor authentication, which may have their justification in production but are not helpful in learning situations. As with all online services, it is advised to carefully consider the use of pseudonym identities and temporary e-mail addresses while trying out the services, while maintaining discipline knowing that they are offered free of charge although real cost occurs for their operation.

9.1 Data pipeline infrastructure

A typical data pipeline involves one or more data sources, reproducible insights generation, and a long-term archive for the results data. In the following, the already available global infrastructure to set up such a pipeline is described. It starts with the acquisition of public data from OpenTransportData, followed by its collection and processing in Renku, and the public archiving in Zenodo. In turn, Zenodo might again serve as one of the input data sources for further pipelines...

9.1.1 OpenTransportData

There are many sources for public data. Well-curated ones with suitable licencing and reasonable long-term availability are open government data (OGD) available from many countries and their subordinate administrative levels. A global view on OGD in selected countries is available from the Open Data Barometer[1] or the older and now archived Global Open Data Index[2]. Individual collections are available from countries like Switzerland[3] or the USA[4] or organisations like the European Union[5] or the United Nations[6].

[1] OGD barometer: `https://opendatabarometer.org/`
[2] Index: `http://index.okfn.org/`
[3] CH data: `https://opendata.swiss/de`
[4] US data: `https://data.gov/`
[5] EU data: `https://data.europa.eu/en`
[6] UN data: `https://data.un.org/`

Complementary to such top-down approaches, community curation of data has led to DBpedia[7] and Wikidata[8]. A third source category are real-time updates mostly from measured data. One can fetch public real-time data on earthquakes[9], consume dweets[10], and inspect live camera feeds[11].

OpenTransportData serves as exemplary input data platform for a global pipeline. It offers access to static and real-time data related to public transportation such as networks, routes, schedules and delays.[12] While it is country-specific, many of the data formats and associated algorithms can also be found in other places, as evidenced by the global Mobility Database referencing almost 2000 static schedules.[13] The OpenTransportData site is among the higher-quality contenders as it is based on a tightly integrated national transportation system and also offers various APIs such as a journey planner. Registration is necessary to obtain an API key for the journey planner and the real-time feed, whereas static data retrieval and some of the other APIs do not require registration as long as temporal request limits are adhered to. The first example translates a coordinates pair to a list of nearby stations, returned as a JSON structure. It should be noted that x refers to the latitude and y to the longitude, in degrees:

```
curl 'http://transport.opendata.ch/v1/locations?x=47.006001
    &y=9.106130'
```

The second example demonstrates retrieval of the live data, including historic live updates dating back to up to a week, in JSON format amounting to hundreds of thousands of lines:

```
# register key first, or use test key for occasional use
testkey=57c5dbbbf1fe4d000100001842c323fa9ff44fbba0b9b925f0c
    052d1
baseurl=https://api.opentransportdata.swiss/gtfsrt2020
curl -H "Content-type: text/xml" -H "Authorization: $testkey
   " $baseurl?format=JSON
```

9.1.2 Renku Lab

Renku Lab[14] is the reference deployment of the open source Renku infrastructure. Renku fosters the collaboration between data scientists and supports

[7] DBpedia: https://www.dbpedia.org/resources/
[8] Wikidata: https://www.wikidata.org/wiki/Wikidata:Main_Page
[9] Earthquakes: https://earthquake.usgs.gov/earthquakes/feed/v1.0/
[10] Dweets: http://dweet.io/see
[11] Insecam: http://www.insecam.org/en/byrating/
[12] OpenTransportData: https://opentransportdata.swiss/en/
[13] Mobility Database: https://database.mobilitydata.org/
[14] Renku website: https://renkulab.io/

reproducible data-driven workflows based on version controlled data, containers and a unique knowledge graph. With Renku, the implications of changes on the input side (i.e. a modified dataset) on the output (i.e. workflow results) become traceable.

In Renku, data and scripts are stored in an integrated Gitlab instance. The Git repository needs to adhere to a specific structure, which is accomplished by running the appropriate Renku commands that create the directories and perform the file modifications as needed. Scripts are typically defined as Jupyter notebooks containing cells of Python code mixed with documentation and reference results.

Experiments are defined as sequential workflows that run the scripts and build a knowledge graph that shows the dependencies and how changes in the code or input data lead to different results in consecutive executions. This way, accidental degradations can be detected and mitigated quickly. The following example shows how to use the `renku` wrapper command to track a script invocation. This commands can be invoked in Renku's integrated Jupyter Lab terminal and will produce a knowledge graph on how modifications to a CSV file were made:

```
# make sure there is a dataset; if not, create it with an
    arbitrary CSV file and register it under a specific name
     <dsname>
renku dataset add --create <dsname> <filename.csv>
renku dataset ls
# inspect the dataset
ls data/<dsname>
# track script invocation
renku run --name <wfname> --no-output rm data/<dsname>/<
    filename.csv>
# check status and save session for later use in case there
     are changes
renku status
renku save
# re-run the recorded invocation for all operations
    affecting a certain file
renku rerun data/<dsname>/<filename.csv>
```

Further commands stress the workflow nature of recorded invocations.

```
# show the workflow
renku workflow show <wfname>
# re-execute a recorded invocation
renku workflow execute <wfname>
# compose a workflow as sequence of multiple existing
    workflows
```

```
renku workflow compose --link-all <wfname> <wf1name> <
    wf2name>
```

Renku also offers an API in addition to the graphical web interface, and a command-line client (Renku Client) to interface with this API from the command line. Alternatively, the API can be used from custom Python applications. Moreover, Renku integrates with public data repositories such as Zenodo and Dataverse in order to import existing datasets.

How to use Renku step by step is well explained in the first-steps tutorial.[15]

9.1.3 Zenodo

Working with data in research settings requires platforms to exchange files and larger datasets. While Git is suitable for data engineering processes, it has comparatively few ways to add publishing information to data or declare relationships between datasets. Moreover, despite technical support for large files, it might not be the best tool to store such files for long-term public access in the first place.

Zenodo[16] is a platform for sharing research data which also archives larger datasets and thus guarantees their availability for further exploration and analysis. Most of the interaction with Zenodo is web-based, such as uploading datasets, describing it with metadata, creating a community and tagging the dataset with the community. Datasets can be imported from Git (and thus also from Renku) and receive a unique publication record in the form of a digital object identifier (DOI). An HTTP API is available to upload datasets but also to search for existing data. An access token must be obtained before being able to use the API for write access such as data deposits. Reading works without such a key, as evidenced by the following example that retrieves the first ten records from a search result and also gives the URLs for navigating to further records: `curl https://zenodo.org/api/records/?q=serverless | json_pp > records.json`. The complete search syntax is described online.[17]

9.2 Distributed applications infrastructure

Building a globally distributed application requires the ability to reach out from one component to another. This is accomplished through a naming and discovery service. Etcd offers such a service. Once the addresses and other metadata of application components are known, communication in the form

[15] One of the Renku tutorials: `https://renku.readthedocs.io/en/latest/tutorials/01_firststeps.html`

[16] Zenodo website: `https://zenodo.org/`

[17] Zenodo search: `https://developers.zenodo.org/#records`

of message delivery between them can start. Dweet offers the appropriate interface for the sending and receiving components.

9.2.1 Etcd Discovery

Distributed systems, and even fully decentralised systems, need a way to bootstrap and identify endpoints. There are many ways to accomplish that in a fault-tolerant way beyond local-only approaches such as environment variables: with a list of meta-servers to maintain a decentralised approach, with registration of records into the DNS of a domain name or informative files placed into the root directory of a web server running on these domain names or finally with a global discovery service. Etcd is a distributed reliable key-value store which for its own purposes, but also for other applications, makes such a discovery service available.[18] In Etcd, key-value pairs can be nested, essentially forming a hierarchy of such entries.

The first step is to request a universally unique identifier (UUID), which is mathematically speaking not entirely unique but sufficient for all practical purposes. The secured HTTP request `curl https://discovery.etcd.io/new?size=5` would create such an identifier allocated for an application with 5 components. Those may be replicas of a single service, or arbitrary components. The response is a single URL that encodes the identifier as a path on the base URL and serves as endpoint for further operations. A typical undashed UUID may thus look like `24f7beedc1e99d5f02ca3cab05124b4e`.

In a subsequent step, naming information about the components are registered. For instance, a component `node1` may want to inform other components of its presence and endpoint. The HTTP write request `curl -X PUT https://discovery.etcd.io/<UUID>/node1 -d value="myip=127.0.0.1"` would register that information. The response in JSON format contains the key as `/<UUID>/node1` and the value as specified in the request.

Hence, a subsequent read request on the identifier makes that information available to other processes or application components:

```
$ curl https://discovery.etcd.io/<UUID>
{"action":"get","node":{"key":"/<UUID>","dir":true,"nodes"
    :[{"key":"/<UUID>/node1","value":"myip=127.0.0.1","
    modifiedIndex":...,"createdIndex":...}],"modifiedIndex"
    :...,"createdIndex":...}}
```

9.2.2 Dweet

Sending and receiving non-confidential messages on arbitrary streaming channels, for instance, in IoT scenarios or other forms of machine-generated data,

[18] Etcd discovery service: `http://discovery.etcd.io/`

is the task of several IoT platforms. They range from standalone or niche platforms to IoT integration capabilities of clouds. A globally usable platform with low entry barrier for occasional needs is Dweet[19].

The interface to publish simple key-value information is, perhaps slightly deviating from protocol conventions, an HTTP GET request of the form `https://dweet.io/dweet/for/<channelname>?k=v[&k2=v2...]`. An HTTP POST mechanism is also available which conforms better to standards and accepts more complex JSON-formatted structured data. Both interfaces respond with a JSON message about the success status and a transaction number. For instance, a script may regularly publish simple information about the available free memory on a machine. It uses a static channel name related to the local identity of the machine, but by combining with the Etcd discovery service introduced beforehand, globally unique names could also be achieved, as follows:

```
freemem=‘LANG=C free -m | grep Mem | awk ’{print $7}’‘
msg="lots%20of%20free%20memory"

while true
do
        curl "https://dweet.io/dweet/for/my::notebook?
            fremem=$freemem&msg=$msg"
        sleep 5
done
```

A consumer script would then retrieve this information. The HTTP connection is kept open to facilitate streaming. Thus, a continuous retrieval could be achieved with the following invocation: `curl https://dweet.io/listen/for/dweets/from/my::notebook`.

Repetition

1. How many stations can be found at the coordinate 47.00° north and 9.00° east?
2. How many datasets with R scripts does Zenodo provide on the topic of rattlesnakes?

[19] Dweet website: `http://dweet.io/`

Solutions

This part of the book contains answers and solutions to the repetition questions and tasks.

2. Concepts: Programming, Data Representation and DataOps

1. Tasks may be in sequential order, connecting the input of one with the output of another; or executing in parallel. Or they may be unrelated, for instance, in a sequence that has other tasks in between. 2. Map invocations are isolated from each other and can be parallelised without side effects or mutual dependencies. The map-reduce computing paradigm relies on this characterisation. 3. It might be a cost-effective and rapid alternative, but it comes with risks such as vendor lock-in and inability to customise the analytics logic.

3. Concepts: Operating Systems

1. The OS attempts to use swap files or swap partitions if these have been set up. If they are not set up or also full, the OS terminates a process. It may be the one that requested additional memory pages. 2. Containerisation typically provides lightweight isolation, with emphasis on fast start-up times. Stronger isolation can be achieved with virtualisation. 3. This website is not registered in the DNS. Either it is registered locally in the hosts files and the server is running, in which case the website is shown. Otherwise, a browser error message is provoked.

4. Concepts: Infrastructure

1. No. This IP address is bound to the local network interface and not to any of the physical network cards (NICs) that would accept traffic from other

computers. 2. OpenAPI is the most widely used language to describe web services, although RAML and other alternatives also exist. 3. Due to the latency sensitivity, cloud computing would be more suitable. Despite clouds not supporting hard realtime constraints, they typically have provisioning models with short response times, contrasting the rather batch-oriented HPC models.

5. Applications and Tools

Shells

1. Logins use the current username by default, but that can be changed with: `ssh Y@X`. 2. The transfer would happen with the following command: `scp Z user@server:`. 3. This command: `ssh user@server screen editor`. 4. At least seven wrapper tools have been introduced at that point: `bash -c`, `ssh`, `sudo`, `screen`, `parallel`, `stdbuf`, `timeout`.

Useful shell tools

1. The TERM signal asks the process to terminate itself, but this request could be overridden so that the process keeps running. The KILL signal is unconditional and always leads to immediate termination. 2. With a construct in the following form: `ps wauxf | grep PROC | grep -v grep`. The last part filters all occurrences of grep in the input. 3. The output of `last` contains one line per login session plus two extra lines at the end. Thus, `echo $((‘last | wc -l‘-2))`.

Shell programming

1. There is no (well-known) command `xyz`. In that case, nothing happens. But it could be a custom-installed program with this name, or a local alias definition, or a shell command. Therefore, when unsure, the nature of the command should be found out first with `type` and `which`. 2. This command would ask to terminate the current shell session. It is however intercepted by the shell for safety reasons. Only a `kill -KILL $$` can override the interception.

Python modules for OS interaction

1. It would print the PID of the just invoked Python interpreter process to standard output, and then terminate. 2. With `os.makedirs(<dir>, exist_ok=True)`.

Package management

1. By running this installation command: `apt-get install vim`. In case the package is already installed, this command may nevertheless lead to an upgrade of the package. 2. Assuming these installations happened via Pip, an appropriate command might be: `pip list | wc -l`.

Container management

1. Not of the OS kernel, but – assuming basic compatibility between kernel and userland – certainly another version of all other components of the OS distribution, or even a different distribution. This could be accomplished and verified best with a versioned run command: `docker run --rm -ti ubuntu:20.10 cat /etc/lsb-release`. 2. This command first requires finding an appropriate image via `docker search pypy`. It should be noted that this search is not resilient and might need several attempts. The output shows that there are lots of choices, but there is also one image plainly called `pypy` hinting at its official nature, and furthermore conveying credibility by having lots of stars assigned to it from users. Therefore, `docker run --rm -ti pypy:latest` might be a good choice.

Data management and version control

1. According to the manual page `man git-merge`, the following six strategies exist as of version 2.39: ORT (Ostensibly Recursive's Twin), Recursive, Resolve, Octopus, Ours, Subtree. 2. First, the repository would have to be cloned: `git clone https://git.savannah.nongnu.org/git/attr.git`. Then, the tool would have to be built. It is evidently an Autotools-based project, hence: `cd attr; ./autogen.sh` followed by `./configure` and `make`. Next, modifications can be applied for instance in `tools/attr.c`, and tested by a rebuild. If everything works, then `git add tools/attr.c` and `git commit -m "<message>"` extends the changes into the local repository clone, with `sudo make install` also making the tool available on the system.

Data processing tools

1. First, a unified phone number format needs to be established, for instance with international prefix. Turned into a regular expression, it can then be grepped out of arbitrary text as follows for notations both with and without spaces in between numbers: `grep -rIP "\+\d{2}\s?\d{2}\s?\d{3}\s?\d{2}\s?\d{2}" .` with the final dot representing the top-most search path. 2. By combining a query of the original size and a conversion, for instance, for PNG

files: `convert -scale $((‘file <in.png> | awk ’{print $5}’‘/2)) <in.png> <out.png>`.

Structured data processing

1. The following pipeline gives a suitable JSON formatting for human consumption based on a provided endpoint URL: `curl -s ’http://.../<file>.json’ | json_pp`. 2. This can be accomplished by a conversion to JSON, as follows: `csvjson <file.csv> | jq ’.[] | select(.A | contains("PRODUCT"))’`, or more directly: `csvgrep -m PRODUCT -c A -d ";" <file.csv>`.

6. Middleware

Programmatic data serving

1. Web browsers enforce CORS. Web services not properly reporting CORS headers do not receive any HTTP requests from the browser, effectively rendering the dynamic functionality useless. 2. The minimal Python code is augmented to listen for HTTP requests and to deliver complete HTML content as responses. This augmentation needs code generation which the streamlit command contains.

File system abstractions and network storage

1. Either using loopmounting with the privileged command `mount -o loop <file> <mountpoint>` or, in case the file system is supported by Fuse2FS, then by calling `fuse2fs <file> <mountpoint>`. 2. SSHFS itself yes; but to automate the mounting, a passphrase-less key is typically used. This means the remote computer can also be reached with an interactive login with the same key, and hence full access to that computer (under the indicated user privileges) becomes possible. In other words, if `sshfs <user>@<server>:<dir> <mountpoint>` works, then `ssh <user>@<server>` also works.

Database interaction and management

1. First, the reason for the slowness needs to be identified. It is caused by many network requests and many small database transactions. A bulk/batch insert operation, if supported by the DBMS, massively speeds up the inserts. 2. Local user authentication refers to the operating system's authoritative information on user identities. Hence, a password authentication is not necessary. Over a network, no such authority exists, and a password must always be supplied.

Message brokers for real-time data processing

1. Polling refers to active checks for new messages with high frequency. It corresponds to a pull mechanism that consumes unnecessary system resources, whereas by using a broker, a process can sleep until it is woken up by the arrival of a new message, corresponding to a push notification. 2. The ZMQ protocol mirrors the underlying TCP protocol. In that, the receipt of each sent packet is acknowledged, and the acknowledgement must be read before another packet can be sent.

Parallel and distributed computing

1. By using spark-submit, the application only needs an empty SparkContext. All resource allocation parameters are set by this wrapper command. In contrast, applications executing without spark-submit must parameterise the SparkContext object regarding local or remote (Spark master) resources and other settings. 2. Not quite. As Spark follows a lazy evaluation approach, it merely records the series of instructions to be run whenever strictly needed. In that case, outputting the dataframe (df.show()) would require the results and run all instructions.

Model serving

1. For the access part, indeed a web server could be used, but it would need logic to upload models, revert to older versions and so forth. The model-serving implementations contain all that. For high scalability, a caching proxy web server could be used in front of a model server. 2. In principle, this is correct. However, the HTTP standard only foresees limited key-value parameters for GET requests, whereas entire request bodies can be supplied with POST requests. Only that way can user-defined data structures required for the inference/prediction process be guaranteed to be expressable in a request.

Data integration

1. Both extractors and taps refer to data sources or read access. Both loaders and targets refer to data sinks or write access. 2. Change management. All changes to the configuration can be verified and traced, and upon misconfiguration, a previous configuration can be easily reinstated.

Workflows and distributed scheduling

1. No. Due to the dependency between both operators, they can only be executed sequentially: `t1 >> t2`. 2. It is a cron expression, meaning that a

job should be executed at midnight (both hour and minute being 0), on all days (day of month, month, and day of week all being irrelevant).

7. Collaboration and Governance Platforms

1. The top-most process would be Jupyter itself. Its child process is the iPython interpreter. And the `ls` process is in turn a child of that one. Each child process is controlled by its parent. 2. There are two commands: first, `podman login <gitlabserver>:5050` as auxiliary step and, second, `podman pull <gitlabserver>:5050/<u>/<project>/<image>`.

8. Execution and Orchestration Platforms

1. No. Floating IPs are assigned by OpenStack to a network proxy that runs outside the VM. Within the VM, only the internal IP address is visible. Software such as web servers needs to be consciously configured to react to public IP addresses or public (fully-qualified) hostnames. 2. No. Only reactive (after-the-fact) autoscaling is supported by the container orchestrator. Applications or helper containers can use their intrinsic knowledge, for instance, upcoming events, to emit proactive autoscaling decisions.

9. Global Infrastructure

1. The stations can be counted with the instruction: `curl 'http://transport.opendata.ch/v1/locations?x=47.00&y=9.00' | jq | grep name | wc -l`. Accordingly there are ten stations nearby. 2. This is a slightly tricky question due to incomplete Zenodo search syntax documentation. With some attempts, the answer can be constructed as follows: `curl "https://zenodo.org/api/records/?q=rattlesnake&file_type=r" | jq ".hits.hits | length"`. Accordingly, there are six such datasets published at the time of writing.

Data Science

Operating Systems and Infrastructure in Data Science

Modern data scientists work with a number of tools and operating system facilities in addition to online platforms. Mastering these in combination to manage their data and to deploy software, models and data as ready-to-use online services as well as to perform data science and analysis tasks is in the focus of Operating Systems and Infrastructure in Data Science.
Readers will come to understand the fundamental concepts of operating systems and to explore plenty of tools in hands-on tasks and thus gradually develop the skills necessary to compose them for programming in the large, an essential capability in their later career.
The book guides students through semester studies, acts as reference knowledge base and aids in acquiring the necessary knowledge, skills and competences especially in self-study settings.
A unique feature of the printed book is the associated access to Edushell, a live environment to practice operating systems and infrastructure tasks.

Print Version:
ISBN 978-3-7281-4167-5

eBook:
ISBN 978-3-7281-4168-2 / DOI 10.3218/4168-2

v/dlf